NAVAL POSTGRADUATE SCHOOL
Monterey, California

15

THESIS

DESIGN, IMPLEMENTATION, BUILDING AND EVALUATION OF A TORUS DOUBLE TRANSITIVE CLOSURE NETWORK OF TRANSPUTERS

by

Jose I. Frazao Sosa
June 1988

Thesis Advisor: Uno R. Kodres

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School — 6b. OFFICE SYMBOL *(If applicable)* 33	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5000	7b. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943- 5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

11. TITLE *(Include Security Classification)*
DESIGN, IMPLEMENTATION, BUILDING AND EVALUATION OF A TORUS DOUBLE TRANSITI
CLOSURE NETWORK OF TRANSPUTERS.

12. PERSONAL AUTHOR(S) Frazao Sosa, Jose I.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Master's Thesis	FROM ____ TO ____	1988 June	168

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the auth and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Transputer, Parallel Processing, Microprocessor Networks, OCCAM Programming Language, Heat Flow in a Plate. (RH)

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Currently the design of highly parallel "supercomputers" is one of the most challenging problems in engineering.
The purpose of this thesis is to describe how the problem was approached in the design, implementation and building of a Torus Double Transitive Closure Network of Microprocessors, using the T414 Transputer as the basic Unit of Computation.
Also compares the performance of the evolved model, from one Transputer to the final stage of sixteen Transputers running in parallel. All the programs and exampler re [illegible] ted in this thesis were implemented in the OCCAM2 Progra [illegible] uage, using the Transputer Development System, D700c, [illegible] release March 1987 compiler version. [illegible]

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. A[illegible] ITY CLASSIFICATION SIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Uno R. Kodres	22b. TELEP[illegible] ude Area Code) (408) 646-2197	22c. OFFICE SYMBOL 52Kr

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—6

Design,Implementation, Building and Evaluation of a Torus Double Transitive Closure Network of Transputers

by

Jose Ignacio Frazao Sosa
Lieutenant Commander, Venezuelan Navy
B.S., Venezuelan Naval Academy,1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

Author : [illegible]
Jose I. Frazao Sosa

Approved by: Uno R. Kodres
Uno R. Kodres, Thesis Advisor

[illegible]
Richard A. Adams, Second Reader

Robert B. McGhee
Robert B. McGhee, Acting Chairman
Department of Computer Science

G E Schacher
Gordon E. Schacher, Dean of
Science and Engineering

ABSTRACT

Currently the design of highly parallel "supercomputers" is one of the most challenging problems in engineering.

The purpose of this thesis is to describe how the problem was approached in the design, implementation and building of a torus double transitive closure network of microprocessors, using the T414 Transputer device as the basic unit of computation.

Also compares the performance of the evolved model, from one Transputer to the final stage of sixteen Transputers running in parallel. All the programs and examples presented in this thesis were implemented in the OCCAM2 Programming Language, using the Transputer Development System, D700c, BETA 2.0 release March 1987 compiler version.

THESIS DISCLAIMER

The reader is cautioned that the computer programs developed in this research may not have been exercised for all cases of interest. While the programs are free of known computational and logical errors, they can not be considered validated. Any application of these programs without additional verification is at risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Instead of attempting to cite each occurrence of a trademark, we list all registered trademarks which appear in this thesis below the firm which holds the trademark.

INMOS Group of Companies, Bristol, UK

- Transputer
- Occam
- INMOS
- IMS T414
- IMS B004 IBM PC add-in board
- IMS B003

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENT

This thesis is dedicated to my wife, Norma for her support and understanding, and our sons Aaron and Guillermo.
To my thesis advisor, Professor Uno R. Kodres and the second reader Major. Richard Adams, I would like to express my thanks for all the confidence and support, given to me at any time when I needed. Finally I would like to thank the technical staff of the Department of Computer Science, especially Russell Whalen, Rosalie Johnson and Mike Williams, for all the aid lent to me during the lab research for this thesis.

I. INTRODUCTION

A. BACKGROUND

1. The AEGIS Modeling Group at the NPS

The research interest of the AEGIS Modeling Group at the NPS, which was created at the late 1970s, is to investigate any possible alternatives to replace the U.S. Navy's mid-1960 design AEGIS COMBAT SYSTEM, and the main focus of attention is the AN/SPY-1A phased array radar processing unit.

Bearing in mind this objective, at present in the Transputer Lab, the main thrust is dedicated to exploring the possibilities that the Transputer, a VLSI microprocessor developed in the United Kingdom by the INMOS corporation. could have in the update process of the AEGIS system currently in use on the U.S. Ticonderoga class (CG-47) Cruisers.

At present the Transputer Lab at the NPS consists of five Zenith PC with B004 Tranputers boards incorporated, two EUROCARD BOXES, one B001 Transputer board, one B002 Transputer board, two B007 Transputer boards for graphics, four B003 Transputer boards with T414 Transputers and two B003 boards with T800 Transputers.

2. Considerations and Terminology about Parallelism

The design of parallel computers is a new frontier in engineering. Since the device and technology is not expected to increase computing power as fast as the increase in demand, novel parallel architectures need to be designed. This design is exciting and important to the future of the computer weapons oriented industry and the national security research projects in this field. Also as with most new frontiers, it is often wild and chaotic due to the little data and methodology to compare the many good designs already in existence.

To help the reader to understand and get a good grasp about parallelism here we have some terminology.
We will start with the basic discussion of terms and concepts in computer architecture. While the readers may be familiar with the terminology, some words were used differently, therefore it is worthwhile to have a concise statement of our use of the word.

We define a processor as a device able to be programmed by a user to act on some data, a procedure as a set of rules that a processor can follow to modify that data, and a process as the execution of the procedure. The Transputer is a microprocessor which includes a processor and special instructions as well as hardware to provide a maximum performance and optimal implementations of the OCCAM model of concurrency and communications.

The OCCAM programming language is the first language to be based upon the concept of parallel, in addition to sequential, execution. It provides automatic asynchronous communication between concurrent processes and is the assembly language of the Transputer, because the Transputer executes the occam programs more or less directly.

A Transputer system is a nonempty set of Transputers including support components to connect them. A parallel Transputer system or Transputer network for short, is a collection of two or more Transputers that is built to work in parallel. A Transputer network is no more powerful, in terms of Turing computable procedures, than conventional computers. We can characterize the networks of Transputers by what they can do efficiently. So we will have two fundamental types of Transputer networks: the special purpose network of Transputers designed for specific applications and the multipurpose Transputer network which is designed to execute most Turing computable procedures efficiently. In this thesis we will refer to a multipurpose Transputer network specifically designed to explore network programming with shared global variables.

The architecture of a Transputer system is the view of the hardware seen by the (systems) programmer. Two machines can have a different architecture if a programmer can see a logical difference between them. A paradigm is a set of architectures based on the same principles.

The Von Neumann paradigm contains almost all multipurpose computers. It is the very well-known paradigm in which a controller, data, memory and (I/O) are sequentially programmed in a fetch-execute cycle, and which contains move, arithmetic, control, I/O, and also logic instructions. The implementation or organization is the block diagram of the computer which shows its memory, processor, I/O and other components, and the realization is the actual hardware of the machine. We will focus on paradigms of parallel computers.

An architecture or paradigm is parameterized if, in the view of the programmer, it has parameters that describe it. Parallel computer architectures may have a parameter, such as the number of processors or Transputers. We can characterize parallel computer architectures as bounded if a parameter such as the number of processors can be efficiently used, and is limited or inductive if the "inefficiency" of the machine follows some reasonable (e.g., sublinear) function of the parameter as it increases inductively (e.g., as we increase the number of processor from n to n + 1). In this thesis we are basically interested in the inductive parallel architectures.

Two other parameters are the number of instruction streams and the number of data streams. A single instruction single data (SISD) stream computer is in general a Von Neumann computer. A single instruction multiple data

(SIMD) stream computer system has one instruction streams (Procedure) simultaneously operating on multiple data streams (data) in separate processors.

A multiple instruction multiple data (MIMD) stream computer has a plurality of different instructions stream, each operating on its own data, we focus on this last type in this thesis.

For our purposes a plurality of procedures that are cooperatively executed on a MIMD Transputer network is a MIMD Transputer network procedure, a MIMD Transputer network process is the execution of a MIMD Transputer network procedure.

In a MIMD Transputer network, the process is clearly a component of a MIMD Transputer network process which is executed in one of the Transputers, where several Transputers cooperate to solve a complex problem or operate independently to solve different problems. We will be concerned with the efficiency of running a simple process in a MIMD Transputer network.

The programers may see a machine that is quite different from the hardware machine, because the functions available to him are augmented or modified by software, microcode or hardware. For example, a MIMD machine may appear to be a SIMD machine by means of the software that implements the synchronization of the processors. When a new machine "architecture" appears due to the use of software, microcode or hardware to change the view of the machine, we call this appearance of the hardware to the programers a virtual architecture. A virtual shared memory system can be created by duplicating information in local memories, so that when a producing process writes a new value into its local memory, the operating system then generates a message to all the consumers of the data. The local memories of each Transputer in the network contain the duplicated data ready to be consumed by each consumer in its local memory. In this way, we have the illusion of working with a Transputer network which physically contains shared memory.

Another interesting concept is the communication, scheduling and synchronization mechanisms between cooperating processes to in a Transputer network. One aspect of this is the granularity of the architecture. A fine granularity architecture is one such that communication, scheduling, or synchronization occurs within an instruction, such as in the fetch-execute cycle of a Von Neumann computer. (e.g., the Transputer OCCAM programming language with its primitives processes send = ! and receive = ?). A coarse granurality architecture implements these operations in terms of instructions as a whole. This definition belongs to the architecture and must not be taken as the granularity concept for the parallel programing. Granularity in parallel programming is a commonly used measure of parallelism, and is an indicator of how much computing each processor can do independently in relation to the time it must expend exchanging information with other processors [HOMO87]. Then a fine-grained procedure spends relatively more time communicating than calculating, in relation to a coarse-grained procedure. A second related aspect is the degree of coupling. A loosely coupled system uses the approach to communicate between simple processors, while a tightly coupled system uses data transfers within the instruction cycle to provide communications between them. Tightly coupled system generally require that each simple process has a fairly extensive knowledge about the other process, while loosely coupled processes may know very little about the other processes. (Knowledge is either an explicit copy of the data that controls a process, or an implicit mechanism such as compiling the procedures from a common source program and running the process in "lock step"). Generally loosely coupled systems require handshaking as in the case of the transputer networks and the tightly coupled system depend on a common system clock to assure the correct completion of a communication.

A third aspect of communication and synchronization is the nature of paths between processors that implement these operations. If cooperating processes have direct wires between them, as in the case of two Transputers connected each other direct operation; if signals pass through other processes, it is indirect (e,g., the case of a network of

Transputers in which for instance the first transputer of a pipeline will send a message to update the data in the local memory of the last tranputers of the pipe); and if signals are handled by additional hardware, then it is switched. For switched communication, scheduling, or synchronization, an interconnection network is used. In this thesis we focus on the indirect case.

B. TRANSPUTER OVERVIEW

1. The Transputer

The Transputer is a computer in a chip - a processor, complete with storage and standard external interfaces. It is a key technological development, because it enables information systems to be designed at a higher level of abstraction than was previously possible (this concept will be discussed later).

Because of its importance, the word "Transputer" has been coined to describe the computer on a chip.
The Transputer focuses special interest on the transfer of information across the chip boundary, rather than on the processing of the information within that boundary. The powerful concept provided by the Transputer links, is an attractive characteristic which makes the Transputer very suitable for building parallel networks [DASP78].

2. Programming Languages

At present exist compilers for Transputers in PASCAL, C, FORTRAN, and ADA (this last will be available for the fourth quarter 1988) but these do not have the capability to exploit the intrinsic parallelism of the Transputer chip and also can not take advantage of the communication model used by the Transputers.

The OCCAM language "understands" parallelism and communication at the very lowest level, allowing the designer to describe and control the use of parallelism in the system. Other languages, regrettably, do not provide the needed facilities; ADA, for example, does not, since its semantics are those of multitasking system (i.e., comprising one or more processes which talk to each other through a shared memory), this implies that a multi-processor ADA system needs a shared global memory. Other languages have equivalent assumptions; any language which provides semaphores, for example, is assuming a shared address space.

OCCAM is a language designed to make the representation and control of parallel systems simple and comprehensible. In addition, it provides most of the facilities that a user of modern block-structured languages like C or PASCAL would expect.

As an example of how the OCCAM language provides for parallelism is that of the transputer processor which provides instruction set support for multitasking and interprocess communication. The model used is that of OCCAM in which the keywords PAR and ALT and the communications operators ? and ! are implemented as instructions. This makes the OCCAM parallelism very fast; a PAR costs around 1 microsecond per component, while the execution time of a matching ? and ! - including all the scheduling needed is about two microseconds [INMOSJ88].

C. THESIS ORGANIZATION

The rest of the chapters of this thesis were organized in the following fashion:

In Chapter II we describe the hardware used during the development of the model, including the Transputer board used to place the I/O handler, which is internal to the PC.

Chapter III presents in a sequential and organized fashion the "growth" of the model from one transputer through sixteen Transputers, which is the final stage of this design, focusing on model evolution, flow of data, and expandability discussion.

In Chapter IV we approach the subject of efficiency related to parallel networks and some key ideas about linear speedup and linear and parallel performance.

Chapter V is a comparative study of the efficiency of the model among the different sizes of the transputer network.

Chapter VI discusses the results obtained in the chapter V, and gives some recommendations about what should be the main goals of the AEGIS Modeling Group from a personal point of view.

II. DESCRIPTION OF HARDWARE USED IN THE NETWORK

A. REALIZATION OF THE TRANSPUTER IMS T414

The IMS T414 was the transputer used in the design of the Transputer network called Torus double transitive closure, It will be depicted for hardware description as well as to gain insight in the functional characteristic of the Transputer chip in general. The T414 integrates a 32-bit microprocessor, four standard transputer communications links, 2K bytes of on-chip RAM, a memory interface and peripheral interfacing on a single chip, using a 1.5 micron CMOS process. For convenience of description, the IMS T414 operation is split into the basic block, shown in the Figure 2.1 [INMOSD86].

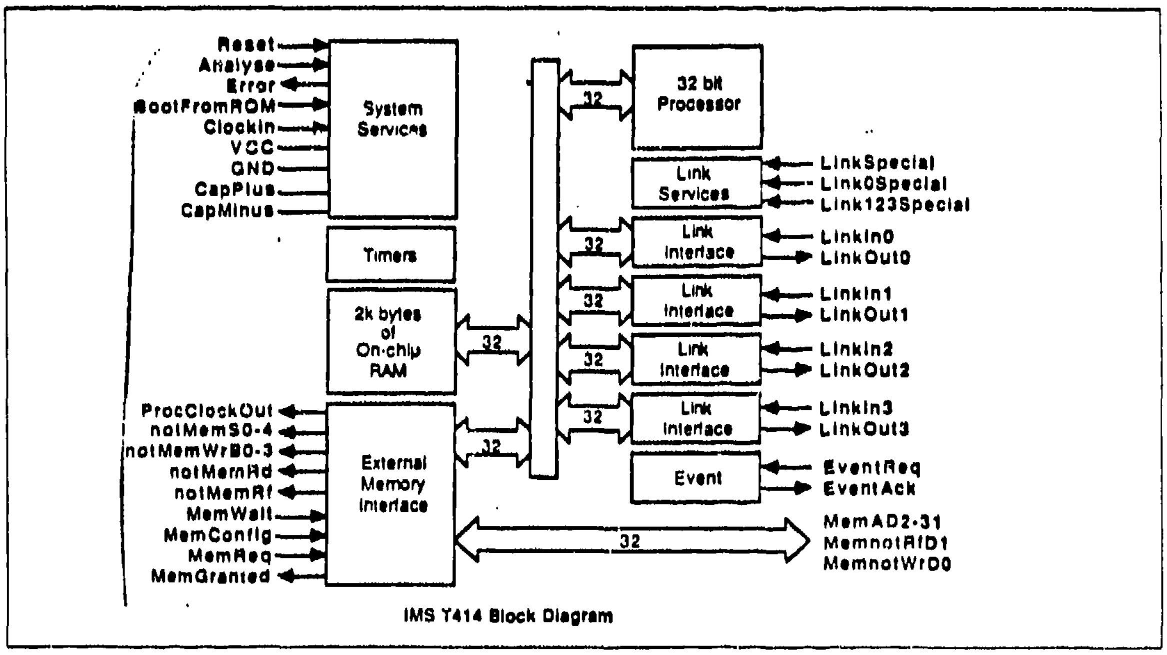


Figure 2.1 IMS T414 Block Diagram

1. The Processor

The 32 bit-processor contains instruction processing logic, instruction and work pointers, and an operand register. It directly accesses the high-speed 2 Kbyte on-chip memory, which can store data or program. Where larger amounts of memory or programs in ROM are required, the processor has access to 4 Gbytes of memory via the External Memory Interface (EMI).

There are only six registers in the transputer, and that is due to the availability of fast on-chip memory. These registers are used in the execution of a sequential process. The small number of registers, together with the simplicity of the instruction set enables the processor to have relatively simple (and fast) data paths and control logic. The six registers are:

The workspace pointer which points to an area of storage where local variables are kept.

The instruction pointer which point to the next instruction to be executed.

The operand register which is used in the formation of instruction operands.

The A, B and C registers which form an evaluation stack.

The Figure 2.2 [INMOSD86], shows these registers.

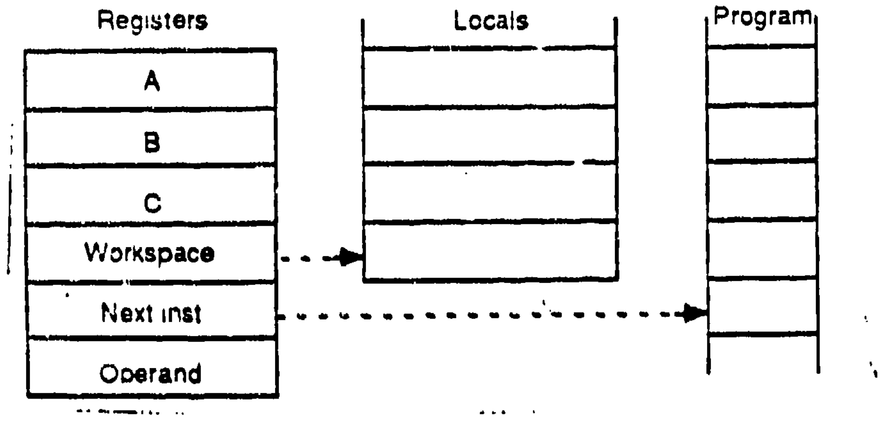


Figure 2.2 Transputer Registers

The A, B and C registers are sources and destinations for most arithmetic and logical operations. Loading a value onto the stack pushes B into C, and A into B, before loading A. Storing a value from A, pops B into A and C into B.

The instruction set has been designed for simple and efficient compilation of high-level languages. All instructions have the same format, designed to give a compact representation of the operations occurring most frequently in programs. Each instruction consists of a single byte divided into two 4-bit fields.

The four most significant bits of the byte are a function code and the four least significant bits are the data value, as shown in Figure 2.3 [INMOSD86].

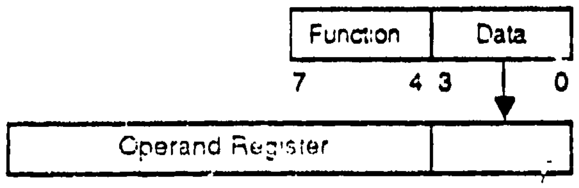


Figure 2.3 Transputer Instruction Format

2. Processes and Concurrency

A process starts, performs a number of actions, and then either stops without completing or terminates complete.

A transputer can run several processes in parallel (concurrently). Processes may be assigned either high or low priority, and there may be any number of each.

The processor has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need of a software kernel.

At any time a concurrent process can be in one of the following states:

Active	-- Being executed
	-- On a list waiting to be executed
Inactive	-- Ready to input
	-- Ready to output
	-- Waiting for a specified period of time

The scheduler operates in such a way that inactive processes do not consume any processor time. It allocates a portion of the processor's time to each processor. The active processes waiting to be executed are held in two linked lists of process workspaces, one for the low priority processes and one for the high priority processes. Each process runs until completion but is descheduled while waiting for communication from another process. In order for several processes to operate in parallel, a low priority process is only allowed to run for a maximum of two time slices (800 microseconds), before it is forcibly descheduled.

The IMS T414 supports two levels of priority. The priority 1 (low priority) processes are executed whenever there are no active priority 0 (high priority) processes. High priority processes are expected to execute for a short time. If one or more high priority processes are able to proceed, then one is selected and runs until it has to wait for communication, a timer input, or until it completes processing. If no process at high priority is able to proceed, but one or more processes at low priority are able to proceed, then one is selected.

Low priority processes are periodically timesliced to provide an even distribution of processor time between computationally intensive tasks [INMOSD86].

3. Communications

Communication between processes is achieved by means of channels. The process communication is point to point, unbuffered and synchronized. As a result, a channel needs no process queue, no message queue and no message buffer.

A channel between two processes executing on the same transputer is implemented by a single shared word in memory; a channel between processes executing on different Transputers is implemented by point to point links. The processor provides a number of operations to support message passing, the most important being input message and output message. The input message and the output message use the address of the channel to determine whether the channel is internal or external. Thus the same instruction sequence can be used for both, allowing a process to be written and compiled without knowledge of where its channels are connected. The communications between two processes is established as follows: The process which is first ready must wait for the second one to be ready.

To be precise, a message is transmitted as a sequence of single byte communications; each byte is transmitted as a start bit followed by a one bit followed by the eight data bits followed by a stop bit. After transmitting a data byte the sender waits until an acknowledge is received; this consists of a start bit followed by a zero bit. The acknowledge signifies both that a process was able to receive the data byte, and that the receiving link is able to receive another byte.

4. Timers

The Transputer has two 32-bit timer clocks which "tick" periodically. The timers provide accurate process timing, allowing processes to deschedule themselves until a specific time. Also they are an excellent tool for programmers to use to evaluate the performance of networks and communication timing.

Two types of timers exist: one for high priority processes and one for low priority processes. The high priority timer is only accessible to high priority processes and is incremented every microsecond, having a full period of about 71 minutes. The low priority timer is only accessible to low priority processes and is incremented every 64 microseconds, and has a full period of about 76 hours.

5. Memory

The 2K bytes of static RAM provide a maximum data rate of 80 MBytes/sec with access for both the processor and links.

The Transputer can also access 4 Gbytes of external memory space. Internal and external memory are part of the same linear address space. Transputer memory is byte addressed, with words aligned on four-byte boundaries. The least significant byte of a word is the lowest addressed byte.

The bits in a byte are numbered 0 to 7, with bit 0 the least significant. In general, wherever a value is treated as a number of component values, the components are numbered in order of increasing numerical significance, with the least significant component numbered 0.

The internal memory starts at #80000000 and extends to #800007FF. User memory begins a #800000048 and is referred to as MemStart.

The reserved area is to implement link and event channels. Figure 2.4 [INMOSD86], on next page shows the memory map of a T414.

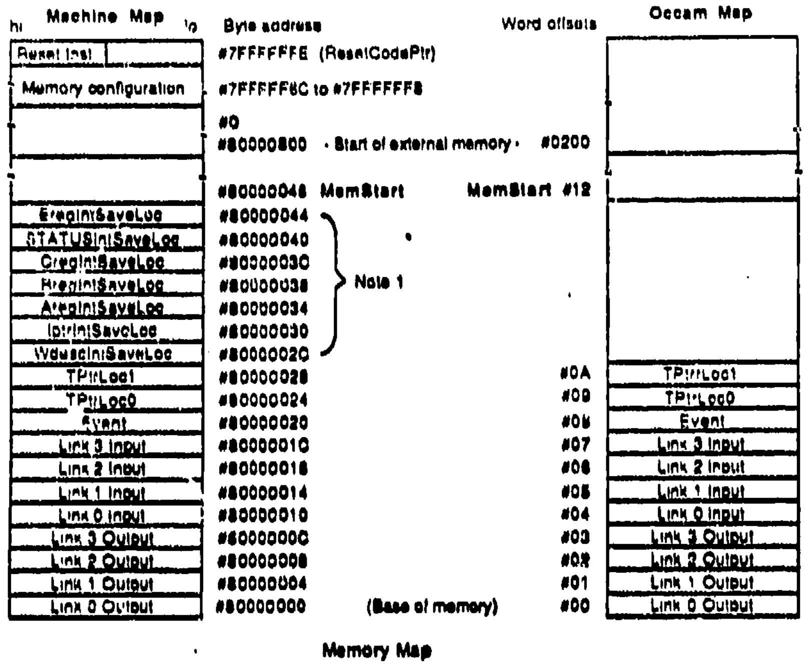


Figure 2.4 Memory Map

6. External Memory interface and Events

The External Memory Interface allows access to a 32-bit address space (4 Gbytes), supporting dynamic and static RAM as well as ROM and EPROM. EMI timing can be configured at Reset to cater to most memory types and speeds, and a program is supplied with the Transputer Development System to aid in this configuration. There are 13 internal configurations which can be selected by a single pin connection. If none are suitable, the user can configure the interface to specific requirements.

EventReq and EventAck provide an asynchronous handshake interface between an external event and an internal process. When an external event takes EventReq high, the external event channel (additional to the external link channels) is made ready to communicate with a process. When both the event channel and the process are ready, the processor takes EventAck high and the process, if waiting, is scheduled. EventAck is removed after EventReq goes low.

Only one process may use the event channel at any given time. If no process requires an event to occur, EventAck will never be taken high.

7. Links

The T414 uses a DMA block transfer mechanism to transfer messages between memory and another Transputer product via the INMOS links. The link interfaces and the processor all operate concurrently, allowing processing to continue while data is being transferred on all of the links. The four links are identical, bi-directional serial and provide synchronization for communication between processors and with the outside world. Each link comprises an input channel and an output channel. A link between two Transputers is implemented by connecting a link interface on one transputer to a link interface in the other transputer.

Every byte of data sent on a link is acknowledged on the input of the same link, thus each signal carries both data and control information. Figure 2.5 shows the Transputer links.

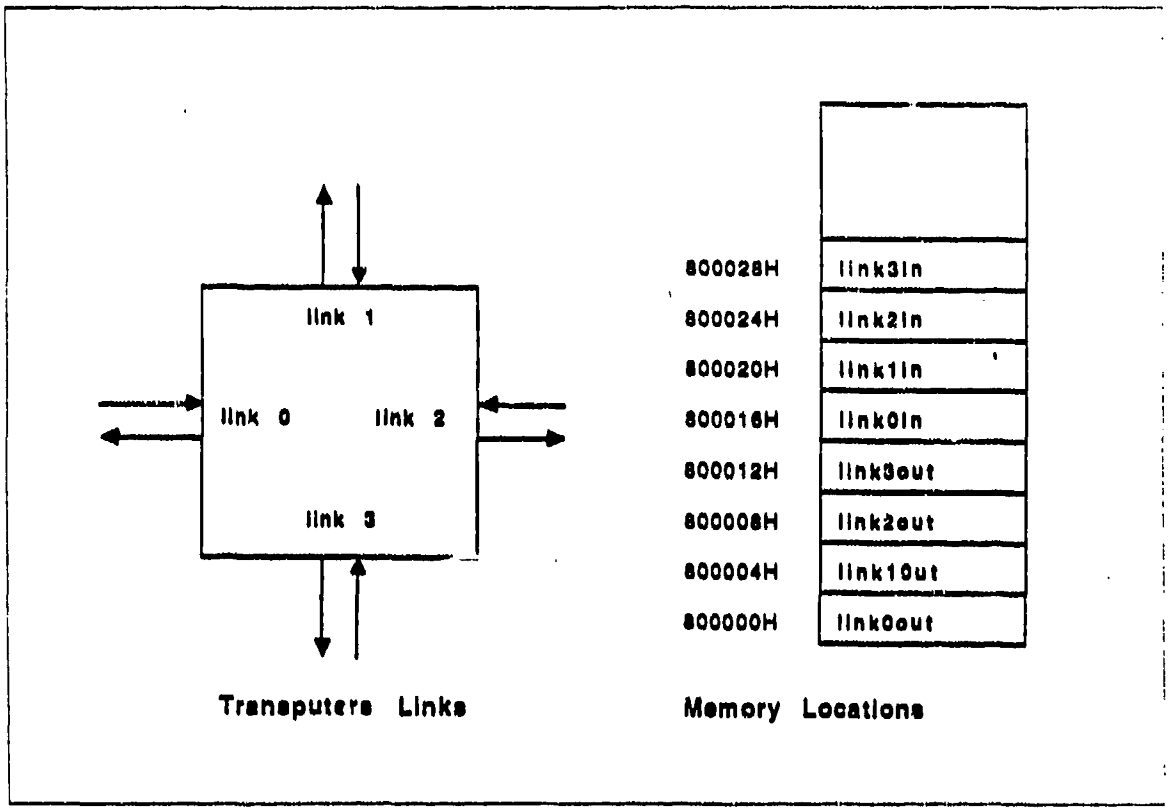


Figure 2.5 The Transputer Links

8. System Services

The System Services include all the necessary logic to initialize and sustain operation of the Transputer. They also include error handling and analysis facilities. They are: Power, CapPlus, CapMinus, ClockIn, Reset, Boot, Peek and Poke, Analyse, and Error.

B. THE B004 IBM PC ADD-IN BOARD

The B004 Transputer board was used to accomplish the function of hold the I/O handler of the transputer network. It is depicted in the following lines.

1. Initial Requirements for the PC Add-In Board

There are three main elements required for the PC board, and those are:

a. A Transputer, with some external RAM

b. The interface to the Personal Computer

c. User controlled devices to allow the board to be used to control other similar boards

Let's talk about the transputer and memory first. The T414 Transputer is a 32-bit processor with a processing capability of 10 MIPS.

For the personal computer add-in board, it was decided to give the user up to 2MBytes external RAM, mapped into the internal RAM of the T414. For this amount of RAM on an IBM form-factor board, dynamic RAM (DRAM) had to be used. Also, a parity check system was implemented.

The communication with the host Personal Computer is handled using the C002 Link Adaptor; this device converts serial link data into byte-wide parallel data, and vice versa. The C002 allows simple interfacing with standard bus architectures, appearing to the host computer as a memory mapped peripheral.

A number of system control signals are also provided which give the user the possibility of connecting a number of Transputer boards to the add-in board via INMOSlinks, allowing the add-in board to control a Transputer network. All signals are software controlled. Figure 2.6 shows the B004 block diagram [INMOSTN11].

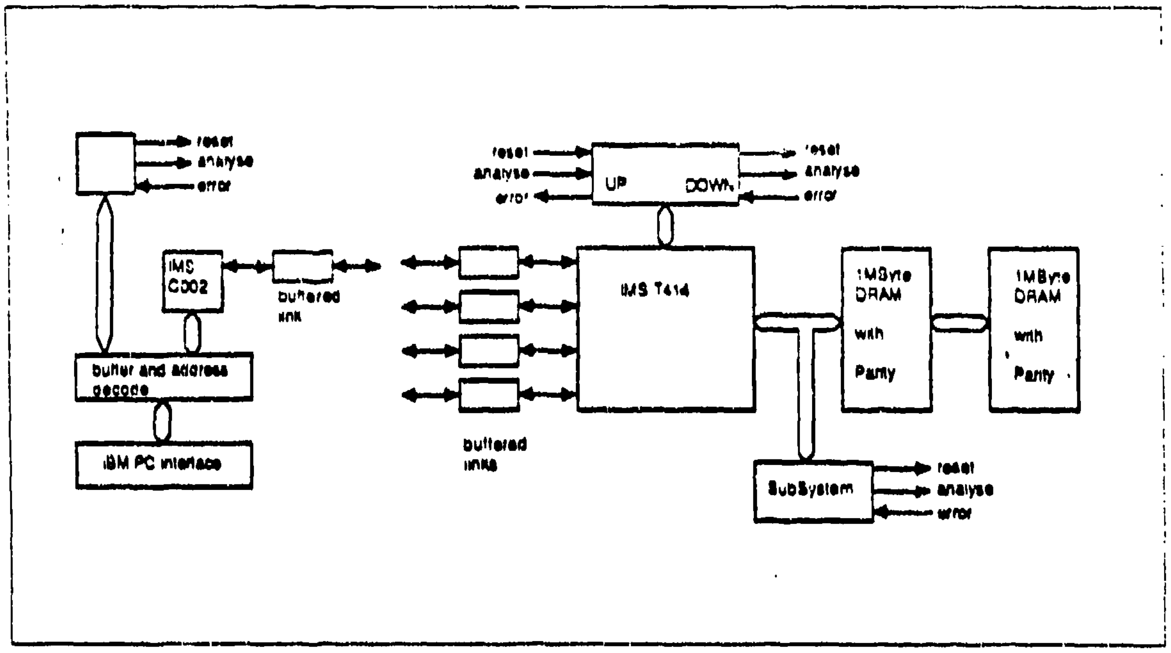


Figure 2.6 Block Diagram of a B004

Because of the Transputer programmable memory interface, we can configure the external memory cycle of the transputer to be any width to suit slow and fast memory.

Also a number of strobes were supplied which can be programmed to give refresh signals to DRAM (automatic refresh over a selectable refresh cycle time can also be chosen). This eliminates the need of timing generators.
The interface with the personal computer is possible due to

the communication between the PC parallel bus and the Transputer via one of the Transputer serial links.

This method was chosen because it maps into the Transputer concept of communications via OCCAM channels, i.e., the host computer appears to be as a process at the end of a channel mapped into one Transputer link. However, that also implies that the Transputer only use a channel to communicate with the host computer.

To make this sort of interface possible, were developed devices which convert parallel data into serial data, and vice versa to match with the channel protocol of the Transputer links.

The aim of the system control functions is to initialize, and analyse errors in an arbitrarily large network of Transputers built with many boards. In particular a B004 board must be able to control many other boards in a rack such as in the EUROCARD BOX.

C. THE B003 BOARD

The IMS B003 evaluation board was the main unit used to build the prototype of the 16-transputer network developed in this thesis.

It comprises four IMS T414 Transputers with 256 kbytes of DRAM in each Transputers. The links provided with the evaluation board allow the Transputer network to be easily extended by connecting with other boards.

This board is capable of processing up to 40 MIPS. The data rate of its links is either 10 or 20 Mbits/ sec. The four Transputers are connected in a ring as shown in Figure 2.7.

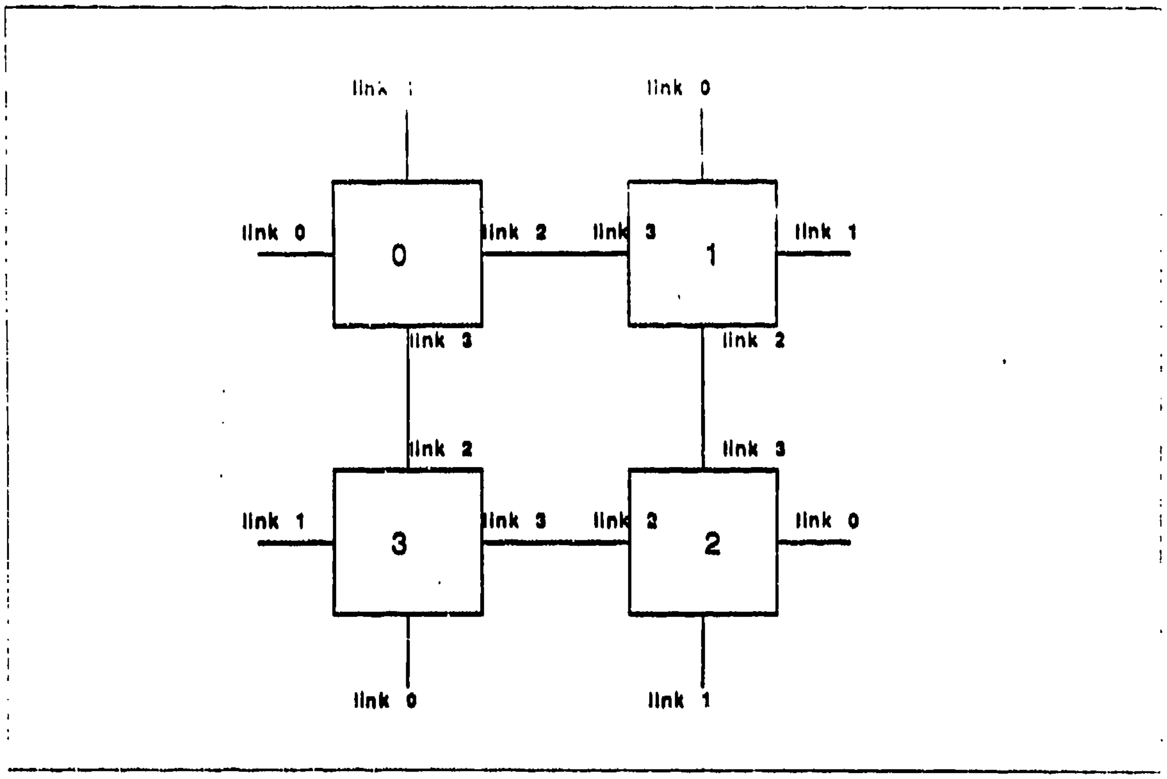


Figure 2.7 The B003 Board

There are two links per Transputer which can be connected externally. Thus each B003 can be connected to four neighbor evaluation boards.

III. DESIGN AND EVOLUTION

A. THE MODELING PROCESS

1. Description of the Problem

The problem chosen was the heat flow problem in a two dimensional plate and how this problem could be solved using globally distributed variables in a transputer network.

This problem was selected because it is representative of many similar types of problems that arise in meteorology, science and engineering.

The heat flow problem in a two dimensional plate is governed by the partial differential equation:

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

with specified initial and boundary conditions.

To find the steady-state temperature distribution in the square plate, one side is maintained at some temperature which is called the hot end temperature, and the other three sides are maintained at 0 degrees (iced bath) as shown in Figure 3.1.

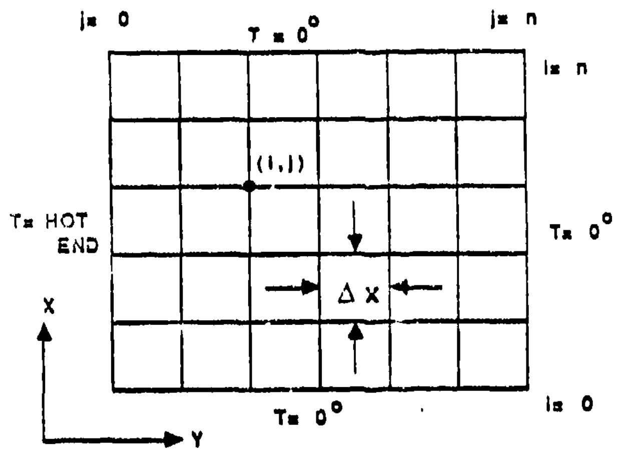


Figure 3.1 Heat Conduction in a Square Plate

All internal points on the grid start also at 0 degrees. Also another element which is present in this equation is the propagation rate W, which is equal to $(1-4r)/r$ where $r = \frac{\Delta t}{\Delta x^2}$

The method of solution is to iterate through all grid points, calculating a better approximation to the temperature at each point (i,j) in turn using the equation :

$$T_{(i,j)} = \frac{((T_{(i,j)} * W) + (T_{(i,j+1)}) + (T_{(i,j-1)}) + (T_{(i+1,j)}) + (T_{(i-1,j)}))}{(4 + w)}$$

As soon as a new value of T is calculated at a point, its previous value is discarded. This is the Gauss-Seidel method of iteration. To start a temperature of 0 degrees is assumed everywhere within the plate. This process of iteration is repeated through all grid points until further iteration would produces, very little change and eventually no change in the computed temperatures. At this moment we have reached the steady-state solution, and we can assert that this is the moment at which the iteration converges, by which we mean, if

$$\lim_{tm \to \infty} T_{(i,j)}(tm + 1) = T_{(i,j)}$$

then our equation satisfies the discretized version of the Laplace's equation.

Our finite difference scheme involves five points, four at time tm and one at the advance time tm + 1= tm + Dt, that allows us to "march forward in time". In this numerical scheme, the temperature at the next time is the average of the four neighboring mesh points at the present time, adjusted by the propagation rate W (relaxation parameter) which is a function of the thermal conductivity coefficient of the material.

2. The Abstract Model

Our abstract model was defined without using a formal specification approach. It can be seen as a black box in which a function operates ruled by the partial differential equation described above. The box provides the solutions to the steady state distribution of temperature in a square plate, with hot end temperature and propagation rate inputs, as shown in Figure 3.2.

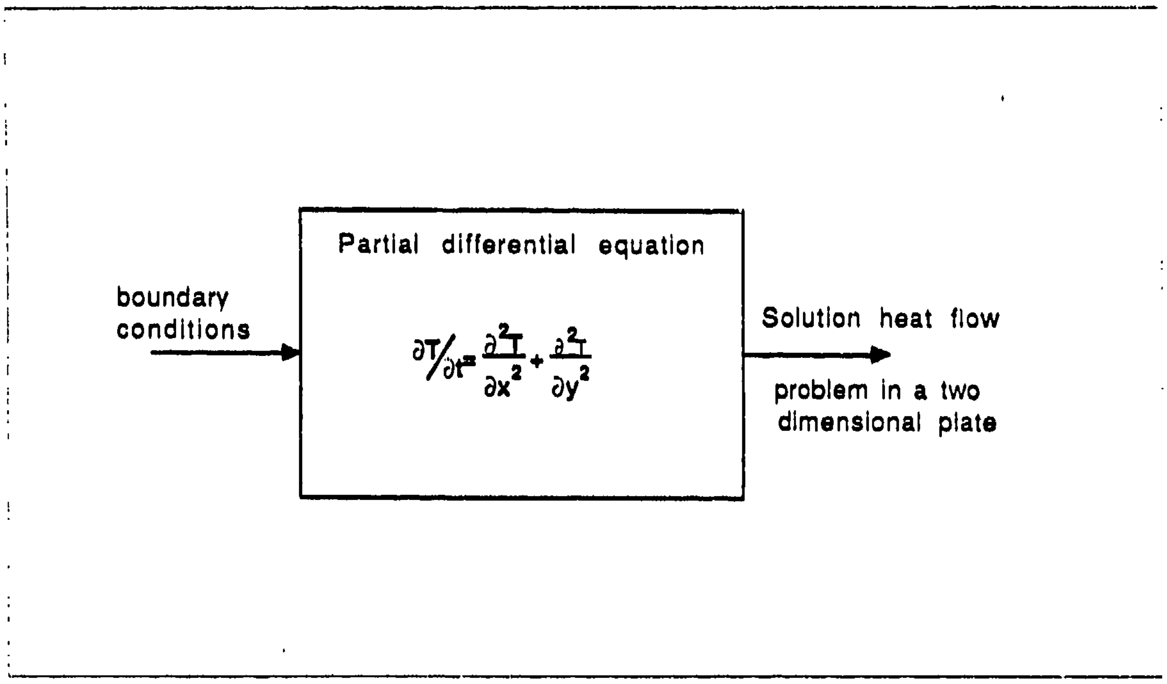

Partial differential equation
boundary conditions
$\partial T/\partial t = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$
Solution heat flow problem in a two dimensional plate

Figure 3.2 Abstract Model

3. The Transformed Computational Model

The Transformed Computational model represents the adaptation of the mathematical model to the facilities supported by the OCCAM programing languages in a modular fashion. This model is shown in the Figure 3.3.

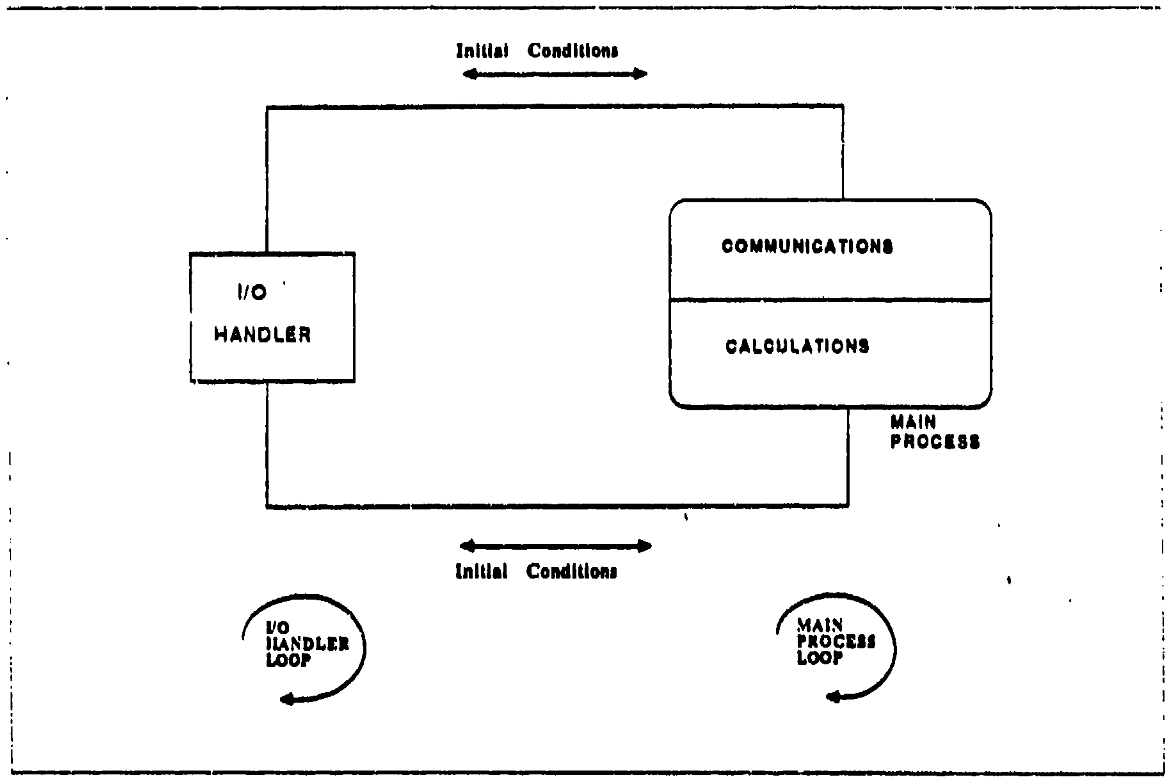


Figure 3.3 The Transformed Computational Model

At the bottom of Figure 3.3 we observe the processes executing. On the left side is located the I/O Handler which is in charge of supply to the Main Procedure with the boundary conditions necessary for the correct operation during each new iteration. On the right side is the Main Procedure box which contains two basic blocks: The Communication Block and The Calculations-Updating Block. The Communication Block is in charge of the maintenance of the interchange of messages with the I/O Handler and eventually with other neighbor Main Procedures.

The Calculations-Updating Block has the functions of calculating the new temperatures for time tm + 1 and also updating the values in the mesh points.

B. NETWORK MODELS AND EVOLUTION

1. Network Classification

We can categorize our network prototype as a MIMD Transputer network, because we have interactions among the n Transputers which comprise the network, due to the fact that all memories streams are derived from the same data space virtually shared by all Transputers. Also this MIMD transputer network is a loosely coupled one, because of the facilities created by the OCCAM programing language.
In particular the input and output messages which use the address of a channel can determine whether an internal or external channel, is being used. Thus the very same instruction allows a process to be written and compiled without having knowledge of where its channels are connected. That is a Transputer does not need to have knowledge about its neighbors to operate properly.
Our final stage will consist of a Transputer network of 16 Transputers connected and operating in parallel to solve the proposed problem of the heat flow in a square plate.

The type of arrangement chosen was a Torus Double Transitive Closure as can be seen on the Figure 3.8.

This type of network is also known as Regular Network [CAWE80] and its main characteristic are the following:

a. The "tree" is a hierarchical structured variation with any processor able to communicate with its superior and its subordinate as well as its two neighbors.

b. If one of the Transputer fails we have redundant paths for single connected failure.

c. The cost of this network is relatively high if we considered its computational power.

d. The modularity and expandability is poor.

e. Performance is very high typically 3 to 5 MIPS, but using the Transputer, we can have higher performance.

2. Model Evolution

Initially we made the set up for one Transputer , but in order to compare the efficiency with a Transputer network, the model was expanded to an array of 2 X 2 , an array of 3 X 3 and the final stage was a 4 X 4 Transputer network.

First let's see the different models which were considered, why they were discarded, and why we choose our final prototype model. The Model I, was a system in which the processes A, B, C, D, E, and F simulated the boundary conditions and the numbered processes achieves the calculations to solve the problem. This model was discarded because for each line of Transputers, it had two Transputers doing nothing but serving to convey the boundary conditions and to extract the final solution of the problem.

Also the communication vertically was very inefficient. The Figure 3.4 depict the model.

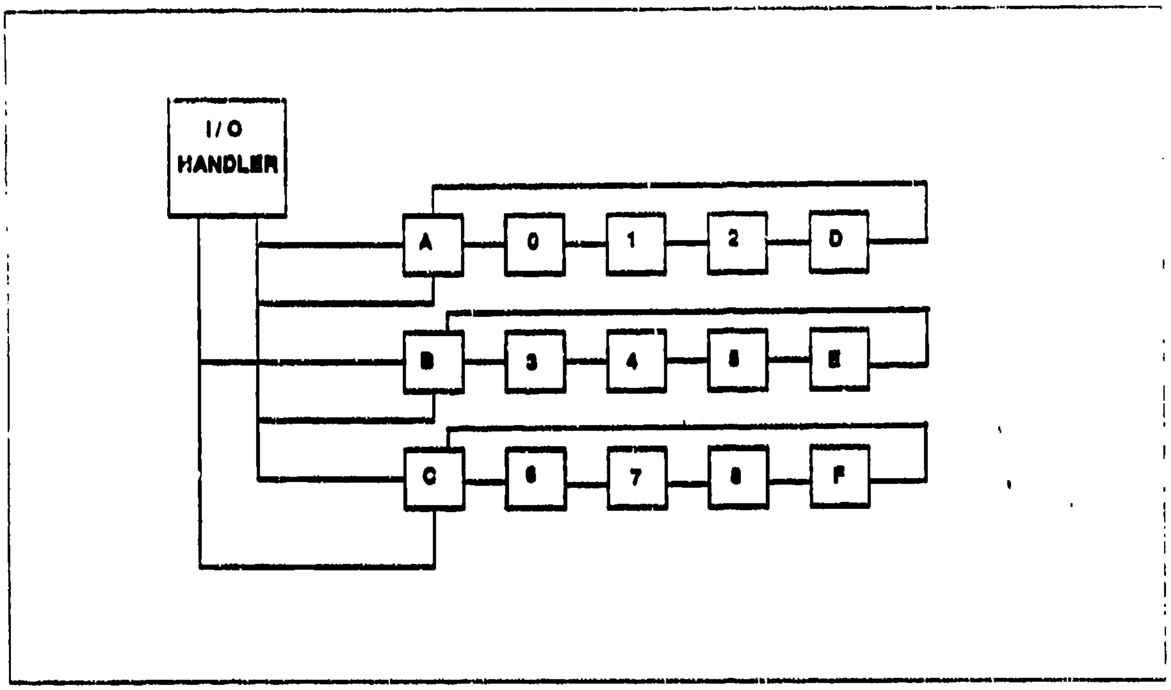


Figure 3.4 Model I

The Model II has the processes A, B, C, and D as senders/receivers of boundary conditions. The main disadvantage of this model is that as we increase size of the network, we will need more Transputers to handle the I/O and boundary conditions passing, this model works well for a small number of Transputers, assuming one is willing to use four Transputers to handle nothing but boundary conditions. This model is shown in Figure 3.5.

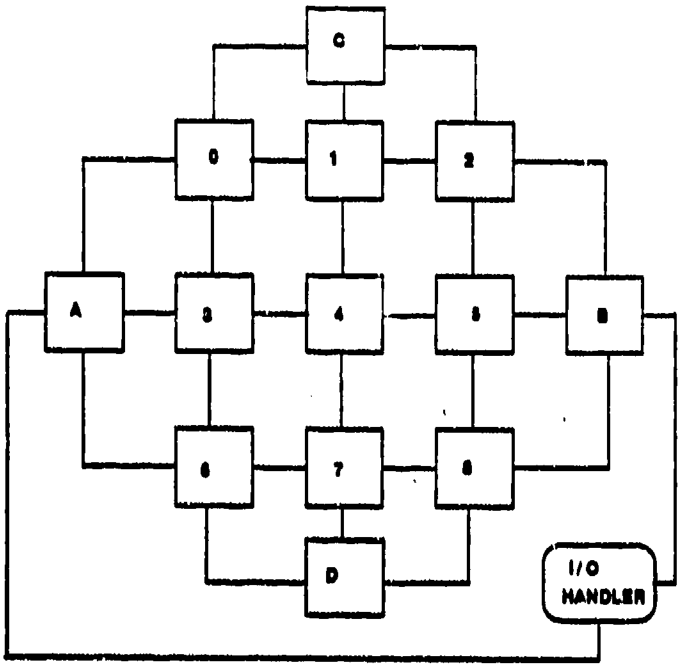


Figure 3.5 Model II

Finally the model we selected, shown in Figure 3.6, is one which handles the boundary conditions better. We use one B003 for the one Transputer network, and make the other three Transputers transparent. The 2 X 2 network used all four tranputers in the board. For the 3 X 3 network we used four B003 using the same idea as for one Transputer in one B003 board, but now making transparent seven Transputers.

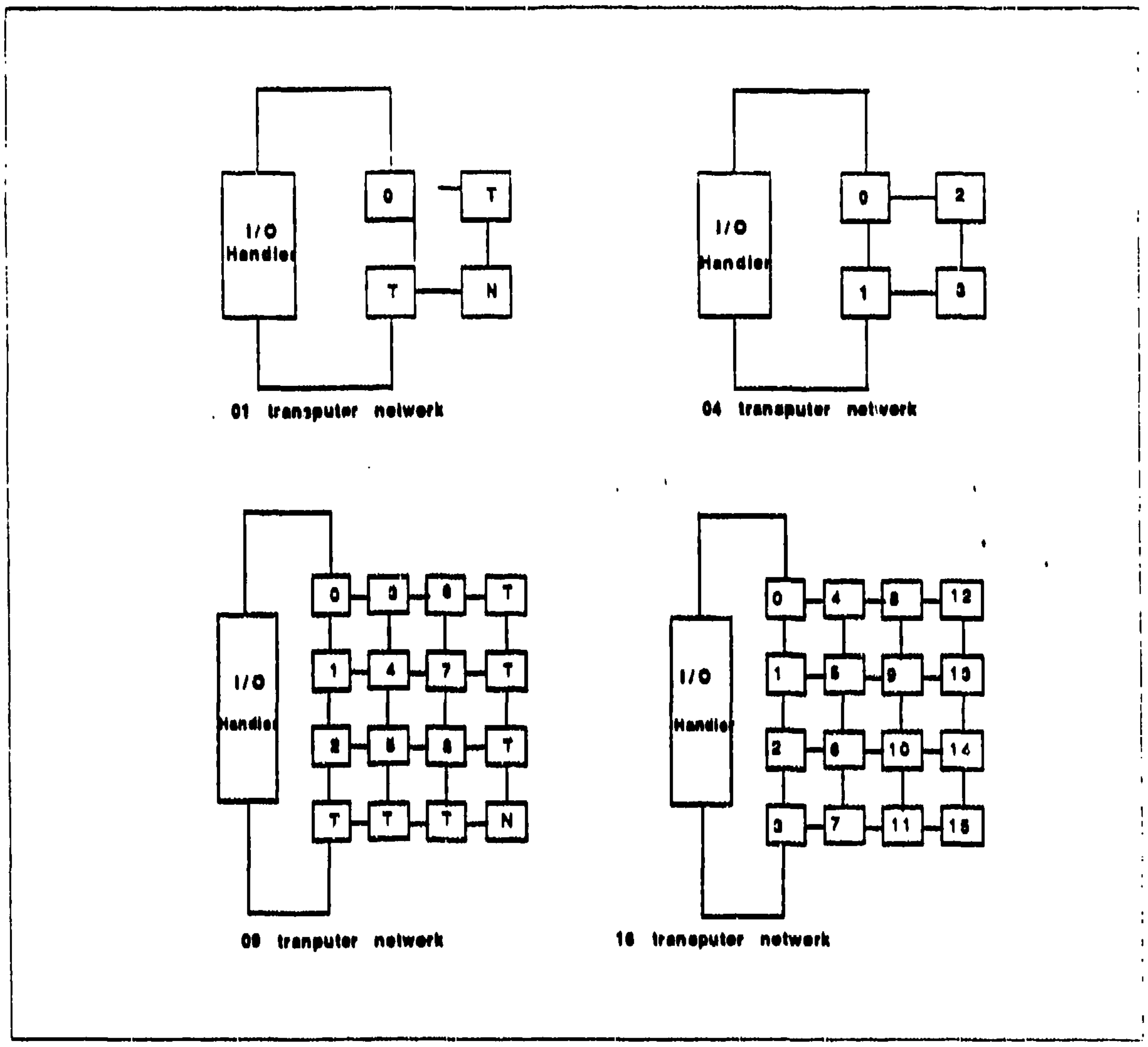


Figure 3.6 Model III in Its Different Sizes

Is interesting to see how the flow of data is achieved in this model. Figure 3.7 shows how the boundary conditions and the start/stop signal are propagated through the network, as well as the data path follow by the solution, when it is sent back to the handler to be displayed on the screen.

In general this model was chosen because it provides the larger Transputer device utilization without have any idle or misemployed Transputers on the four and sixteen Transputer networks, and also because its symmetry permits a more even distribution of the communication load in the network.

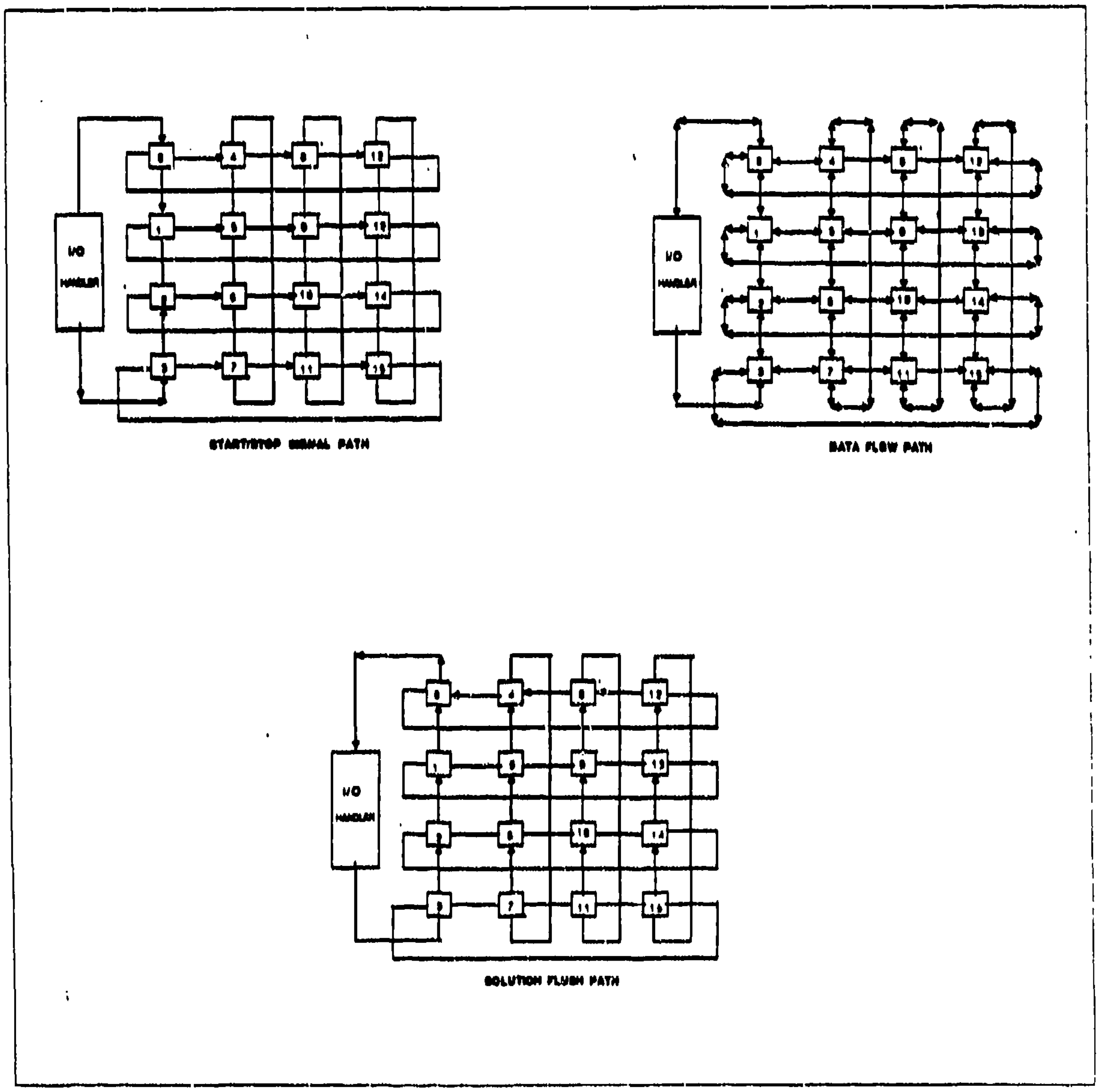


Figure 3.7 Data Flow in the Network

3. The 16 Transputer network prototype

On Figure 3.8 we can observe the 16 transputer network with all its connected links, including these which communicates to the I/O handler. The programs for each one of the transputer networks are contained in the Appendixes A, B, C, and D; the implementation of the modules are those programs, and they will be discussed in the next chapter at the paragraph, Maximization of Software Performance.

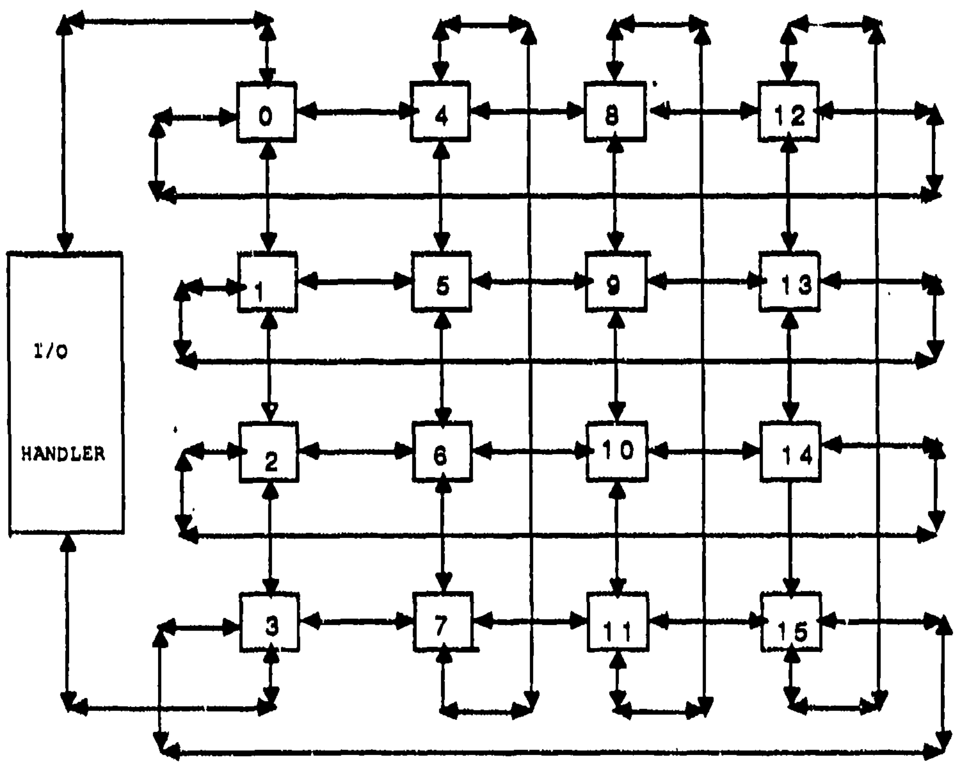


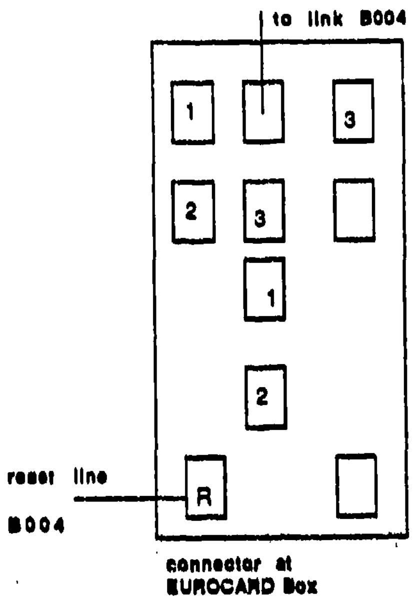
Figure 3.8 16 Transputer Network Prototype

4. Expandability of the Model

This transputer network can be expanded easily using the series, (2 + n)^2 in which n is 0, 2, 4, 6, 8,..... this allows the construction of Transputer networks utilizing all the Transputers available on the B003 boards which is not the case if we get n= odd, then the Transputers that are left over must be made transparent in order to run the network. This practice however, makes the placement of the channels a job tedious and error prone.

Appendix E is contains an expandable placement of Transputer channels following the above series for n even. Thus we can easily place with just change a number, networks of 16, 36, 64, 100 ..... Transputers [INMOSTN13].
The way in which the external links were connected, including the links that joined the different B003 boards on the EUROCARD box, is displayed on Figure 3.9 for 01 and 04 Transputer networks, and in Figure 3.10 for 09 and 16 Transputer networks.

The connecting box(es) shows the connections between the various B003 boards which make up the Transputer network.



To set up the external links, just match up the numbers using the twisted cable, provided with the boards.

Figure 3.9 01 and 04 Transputer Networks Connections

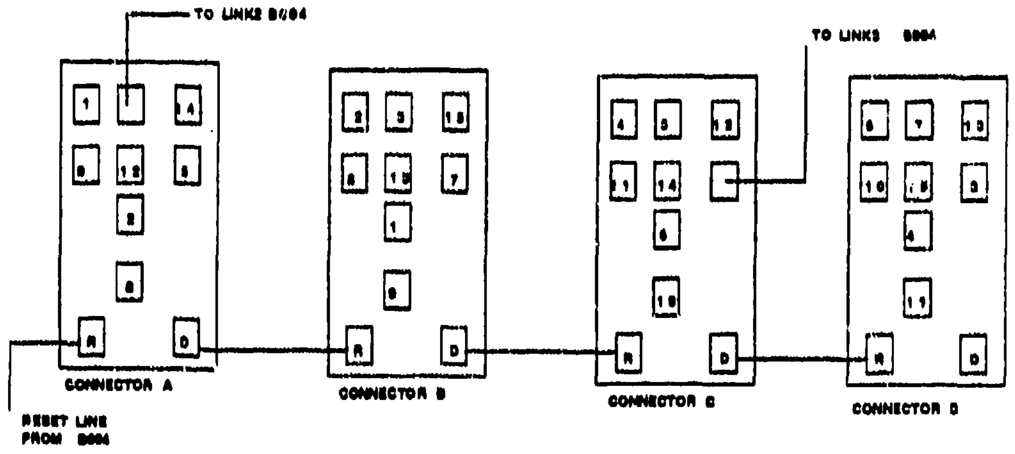


Figure 3.10 09 and 16 Transputer Networks Connections

IV. EFFICIENCY CONSIDERATIONS

A. INTRODUCTION

1. Generalities

Two related aspects of a parallel computer that affect run-time efficiency are the speed of computation and the speed of communication. The first relates to the design of the processor, its instruction set, and its organization (such as the use of a cache and pipelining) and its realization (such as the speed of its transistors). The second relates to the interconnection network, the scheduling of its resources and the routing of information through it. This second aspect is less understood, and is the one in which different paradigms of parallel computers differ most. We focus on these two aspects, and propose that our application be characterized by its communication requirements. Applications with similar communications requirements can be grouped together. For instance a pattern recognition edge-detection problem can also be put in a Transputer network mesh structure and our two dimensional heat problem also can be put in a Transputer network mesh structure. These two mesh structure problems have radically different computational requirements, but have the same communication requirements. We can study such network topology from the point of view of how well it handles a related class of Transputer network procedures.

We agree that a large Transputer network can be built to solve large problems, then we will submodel that problem into two models; one model considers how a very large Transputer network can be built, and how a MIMD Transputer network process can be expanded within it, to determine whether doubling the number of Transputers assigned to that problem will speed up its execution by a factor of two. We might get linear speedup if that were true. (This ideal situation is not easy to achieve, unfortunately). Those results about linear speedup of Transputer network procedures are very important since we need good procedures for Transputer networks. The other model which is complement of the first assumes that the problem size will remain fixed and the machine will be larger and larger inductively. That is, the problem may be run on one Transputer, and the machine might be expanded from one to sixteen Transputers, and we will consider the efficiency of running the problem on the same one Transputer.

This model is easier to study, since rather simple and general statements can be made on it.

It is quite useful in understanding the overall model, since expanding a Transputer network system to solve a bigger problem can be done by fixing the problem and expanding the machine first, then expanding the problem to fill the machine.

In this thesis we will devote correspondingly more time to studying the model which shows how a given Transputer network process can be expanded within a large machine to determine if increasing the number of processors assigned, we can get linear speedup, also some reference and results related to how the fixed sized problem behaves when the Transputer network system in which it runs is expanded inductively [LIMI87].

2. Terminology and Concepts

We want a suitable set of definitions to evaluate the quality of our architecture. Because of that, a notion of "energy" is given besides the traditional concepts used in engineering for the efficiency study.

a. Power and Energy

The computational energy for a process is the product of the computational power (bit rate able to be generated by the hardware of the Transputer) and the time the hardware is needed, where the computational power includes all the output necessary to run the processes and the time is the product of the length of the clock cycle or in other words is the time required for computation and communication.

To clarify those concepts let's see an example, suppose we have a network with four Transputers like the case of the networks that can be implemented using a IMS B003, then if each Transputer has a computational power of

10 MIPS we can then assert that our network comprised of four Transputers will have a computational power of 40 MIPS. Therefore if we have a module running a process, we can use N identical modules to execute the same process (as N Transputers) but having available N times the computational power of one module.

When we expand our Transputer network in an inductive way, we call each Transputer that we add a unit of computational power (UCPs).

In the evolution of our 16 Transputer network prototype, we pass through the 3 X 3 network which is assembled using 4 boards B003, then in this topology we find a special kind of Transputer which is transparent or a neutral unit. It does not compute and only has the task of moving data in and out of the network or simply doing nothing as the Transputer located at the right-lower corner. These modules cannot be classified as UCPs, so we call them blocked UCPs, and these will be considered when we evaluate the Transputer network in the next chapter. We also take these into account when we measure the total amount of computational energy necessary to run a Transputer network process. The Time is also an interesting concept, and it includes all the components of the time needed to execute a Transputer network process. We will break the time in two main blocks; the communication and the calculation time. These two blocks are very well defined in our Transformed

Computational Model from Chapter III and also they can be seen on any of the programs from the Appendices.

b. Efficiency

The usefulness of a computer is indicated by the efficiency it exhibits in the execution of processes on it. This is the obvious definition for efficiency; now we will define relative efficiency as well as the concept of equivalent process necessary to understand the relative efficiency. Later a relation between relative efficiency and input computational energies will be stated.

The relative efficiency of two computer systems executing equivalent processes is defined as the ratio of the efficiencies of the two systems in executing the process, where two processes are equivalent if they provide the same outputs when given the same inputs, (which is clearly our case in the network). Therefore we can define the efficiency of a computer system in executing a process as the ratio of input computational energy (ability to generate bits from the modules) to the output computational energy (information of theoretic bits produced by a module).

From this definition we can state that :

the relative efficiency of two computer systems executing equivalent processes is inversely proportional to the ratio of input computational energies of the two computers [LIMI87].

3. Applications of Efficiency Analysis

So far the reader probably has some doubts about the concept of efficiency and that it is critical for the analysis that we present in the next chapter. Thus to bring some light, let's use it to analyze some issues to show its utility.

First of all, we will consider the simple idea of serial-parallel conversion, which leads to the notion of speedup. Before we do that we will classify the efficiency analysis in two types; first order analysis which ignores communication and control, focuses on computation, and the second-order analysis which considers all these factors.
Then the analysis that we use to determine if a procedure is capable of linear speedup, may be a first-order analysis and to understand the real world we will need to apply a second-order analysis. In Figure 4.1 we can observe the classical comparison between parallel and pipelined processors, this is a simple notion which has been manipulated by theorists for many years.

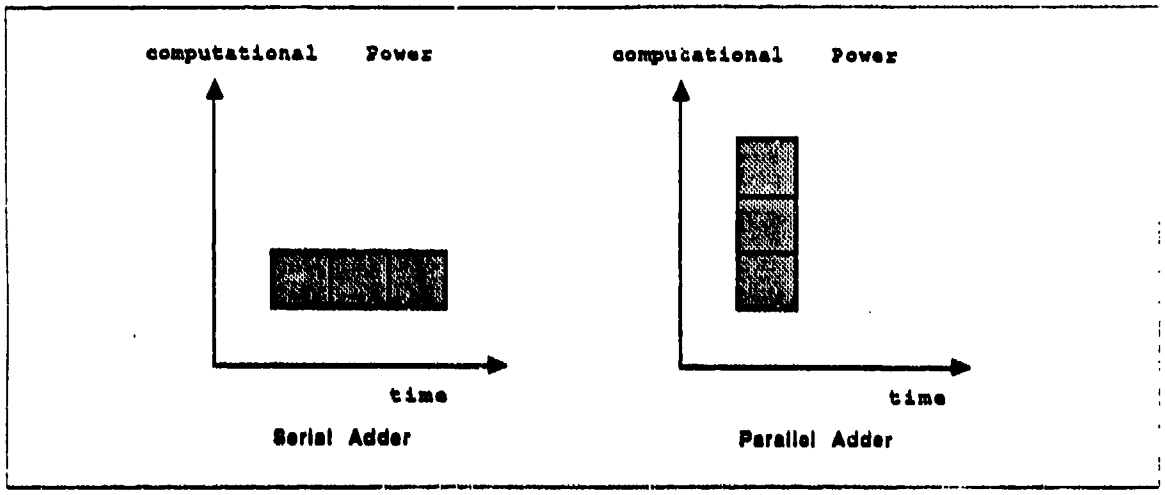


Figure 4.1 Energy for a Serial and Parallel Adder

If we examine the relative efficiencies of a serial and a parallel adder (Fig. 4.1), in which the computational power of the adder cell is much more greater than that of the control and communication circuitry that support the adders (i.e., the calculations are more time consuming than the communications), therefore we will ignore these factors (first-order analysis). The energy for a 3 bit serial adder and for a 3 bit parallel adder is shown in the Figure, in the serial adder we have one unit of hardware used for three units of times and in the parallel adder we have three units of hardware being used for one unit of time, then clearly the areas are the same and so are the relative efficiencies. This simple procedure shows the notion of linear speedup. If the number of UCPs is multiplied by N then the time to execute the procedure is reduced to 1/ Nth, or the speed is increased by N.

It should be noted that linear speedup is equivalent to constant computational energy. This, type of analysis will be used extensively in the next chapter, when we perform the comparative evaluation of the different networks. The results are misleading in some architectures because it does not consider the changes in computational energy due to the communication and control. Nevertheless the analysis carried out on the different prototypes was of the type second-order, because the communication time was include in the total time.

From the notion of linear speedup and conversion of serial to parallel we can realize about the secondary importance of the speed as figure of merit in a topology. A parameterized architecture based on a single procedure as addition is capable of considerable speedup. For instance 12-bit add can be done one bit at a time in 12 time steps, or 12 bits at a time in one time step. Within limits, it is possible to squeeze the time dimension of an energy area as the power dimension is increased to get constant area.

The degree to which parallelism can be exploited to get speed depends on the amount of data to be processed. However the limit to the speedup is given by the smallest size of the unit of computational power (i.e., indivisible), and this is the fundamental idea why the researchers are interested in fine grain rather than large grain parallelism.

Ultimately we can not go further than a Turing machine. Within these limits, time can be traded against power. Thus the speedup is not a fundamental figure of merit for a parallel architecture. The more fundamental figure of merit in a parallel architecture is the efficiency.

B. MAXIMIZATION OF THE TRANSPUTER NETWORK

1. Generalities

This section will describe how to obtain better performance from a Transputer network (array type). However only very general guidelines can be given, because this area is still on active research and our solutions tend to be specific to our problem.

2. Maximizing link performance

The Transputer link is an autonomous DMA engine capable of sustaining a bi-directional data rate of 20 Mbits/sec. However in our prototype we are using 10 Mbit/sec as the common data rate. The higher rates can be used without seriously degrading the performance of the processors. To achieve a maximum link throughput the system links and the processor must be kept as busy as possible.

Following are some suggestions for achieving the maximum throughput:

a. Decoupling communication and computation

To avoid the links waiting for the processor or vice versa, link communication should be decoupled from computation. For example, it is inefficient to have code like the following :

```
SEQ
  in ? data
  compute(data)
  out ! data
```

because we are forcing the Transputer to perform one action at a time, as inputting, computing, and outputting. The solution is doing the three things at the same time using a couple of buffers into a parallel construct:

```
PAR
  buffer(in, a)
  compute (a, b)
  buffer(b, out)
```

b. Gather together all the communications processes

This can be seen in the communication blocks of the diverse designed prototypes. The communication process must also be wrapped into a PAR construct. If possible, is also recommended to put this PAR package inside a PRI PAR running first or at high priority, the communications package.

c. Large link Transfers

When we set up a transfer down a link, the set up itself takes about 1 microsecond. Once the transfer is initiated, it will proceed autonomously from the processor, consuming typically 4 processor cycles every 4 microseconds. Thus the idea is to keep the message as long as possible. However, long data transfers also increase latency when data must be transferred, which occurred in our case for the 16 Transputer network prototype. To solve the problem we used the optimal message length in all the topologies developed, including the final model of 16 Transputers, which used between 10 and 100 bytes [SIHA88].

d. How the boundaries were passed in the network

The problem of the boundaries exchange was approached in the following manner: The basic idea was to send and receive by all the channels available, and if the information (boundary) was not necessary, we just do not use it. It may appear inefficient but for purposes of creating homogeneous processes, we favored this option. This gives a uniform communications package, allowing a better measure of the performance to be obtained. The boundaries were one dimensional linear arrays with a maximum length of 24 integers (one Transputer network) and a minimum length of 6 integers.

Figure 4.2 shows how this sequence of events happens. Once the communications are achieved the different boundaries are stored in linear arrays called dummies, then the processes decide whether to use them or not.

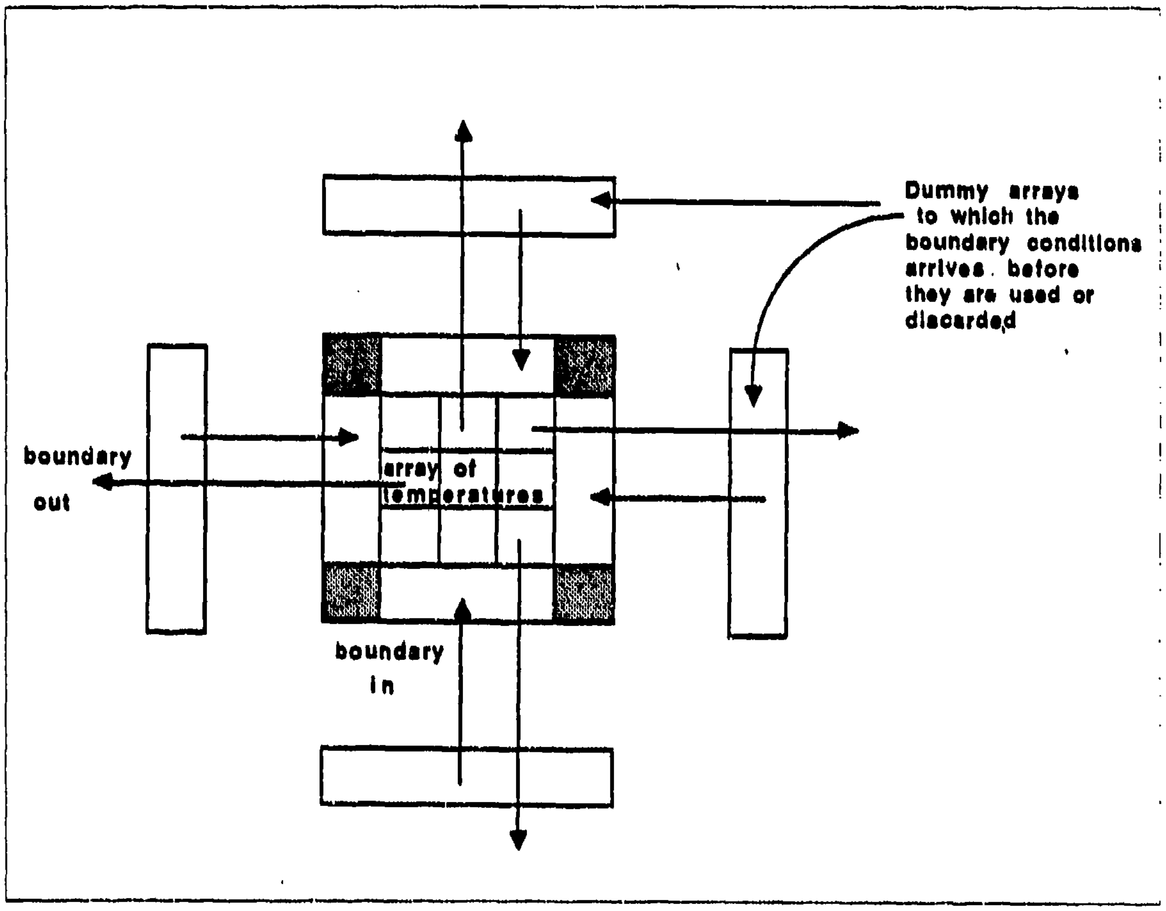

Dummy arrays to which the boundary conditions arrives before they are used or discarded
boundary out
array of temperatures
boundary in

Figure 4.2 Boundary Exchange

C. MODULARITY OF THE SYSTEM

The modularity of this type of system is poor [CAWE80]. The main issues that conspire again the modularity of each of the procedures were the routing code for the start/stop signal and the routing code to extract the final information from the network. Even considering this little difference in the implementation of each module, we still preserve the data structures for the Communication and Calculation Block identical. We call these two blocks the main data structure, which allows us to see the Transputer network as a system with virtual shared memory by duplicating the information in each main data structure which is in turn a block of memory on each Transputer.

The routing codes are different, however because most of the Transputers in the network have to perform a different job to assure the transmission of the start/stop signal and to flush the results out of the network.
For instance Transputer number 0 which is at the upper left corner, has to receive and send to the I/O Handler 15 arrays of temperatures plus its own array, in contrast to Transputer number 3 which is at the lower left corner, and which only has to send up its own array the moment after the reception of the stop signal.

V. COMPARISON OF NETWORK PERFORMANCE

The main reason to build parallel computers is to be able to solve larger problems or to solve the problem faster.

This chapter focuses on the central theme of this thesis. We have described so far a parallel computer (Transputer network) prototype which has been implemented in an inductive fashion. Briefly, an inductive architecture is one that can execute a number of jobs proportional to the number of processors, and the energy needed for each job is proportional to a sublinear function of the total number of processors. Thus a relatively large process, as the one used in this thesis (heat flow problem) whose procedure exhibits linear or nearly linear speedup, can run efficiently on the whole network if it has an inductive architecture.

It is also convenient to comment that the experimental results obtained from the different Transputer networks were conducted using off-chip memory data. This provides the worst case evaluation and all the results are under the same general conditions.

A. ARE WE USING AN INDUCTIVE ARCHITECTURE ?

After the above lines and before get into the efficiency subject, we think it is good to verify this point.

A computer architecture is inductive if:

1. There is a basis architecture, and all architectures use only the components that are units of the basis. For us that is certainly true, since the basis architecture is represented by only one Transputer, and the other architectures contain nothing but the same UCP, which is the Transputer.

2. There is an induction mechanism that can expand an architecture from N UCPs to N+1 UCPs. That also can be seen in Figure 5.1, in which we see the basis architecture on the left and the expanded architecture on the right for a simple N by N mesh. The induction mechanism simply adds Transputers around the perimeter of the mesh to increase the number of UCPs from N^2 to (N+ 1)^2.

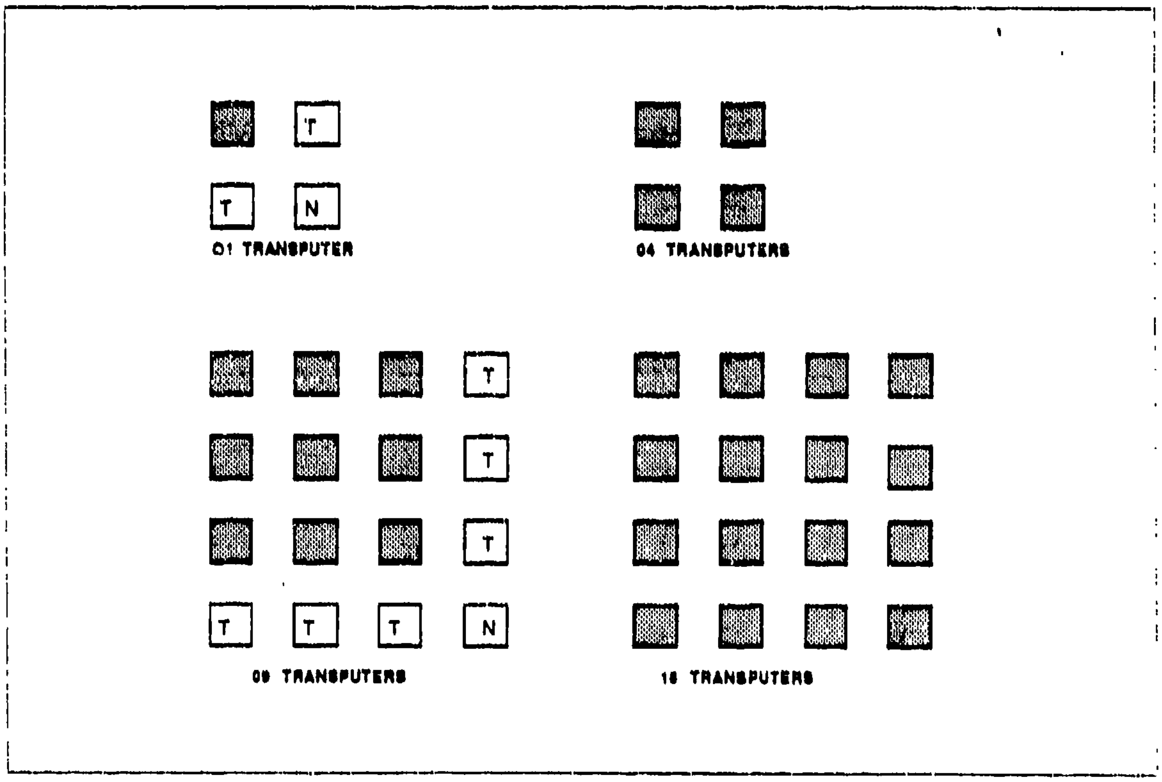


Figure 5.1 The Inductive Mesh Architecture

Therefore we can assert that our expanded model is an inductive architecture.

B. EFFICIENCY EVALUATION

In general the efficiency can be increased by using a better procedure, a faster technology or processor. In this thesis, using the inductive property of our architecture, we will not change the technology, procedure or processor, but we will use a variable number of identical processors (Transputers). What we have done in this evolution, or better, induction of the basis model is to fix the size of the problem. That is, we are solving an array of 24 by 24 elements and executing it on more UCPs or Transputers; our goal is to show how this Transputer network process runs, without seriously decreasing in efficiency. Then from our experimental results we can see in Figure 5.2 a picture, which is pretty much the same as the one used to describe the linear speedup concept in Chapter IV; the sizes of the UCPs differ a bit from the original basis, but this is due to the fact that we are using a second-order analysis in which the communication and control overhead is considered, and of course larger than for only one processor running the same process. In this Figure on the left, the area of the rectangle is the energy to execute the process in one Transputer, and on the right and the bottom we can observe the same for the other inductive architectures.

In Table 5.1 we have a summary of the results.
We can observe that the computational power of each network is incremented as expected by factor of 4, 9, and 16 in relation to the value of the network of one Transputer.

TABLE 5.1

PROTOTYPE ENERGY RESULTS

time	computational power	# of Transputers
30.82 sec	310,519 bit/sec	01
05.74 sec	1,400,102 bit/sec	04
02.33 sec	2,829,484 bit/sec	09
01.08 sec	4,943,449 bit/sec	16

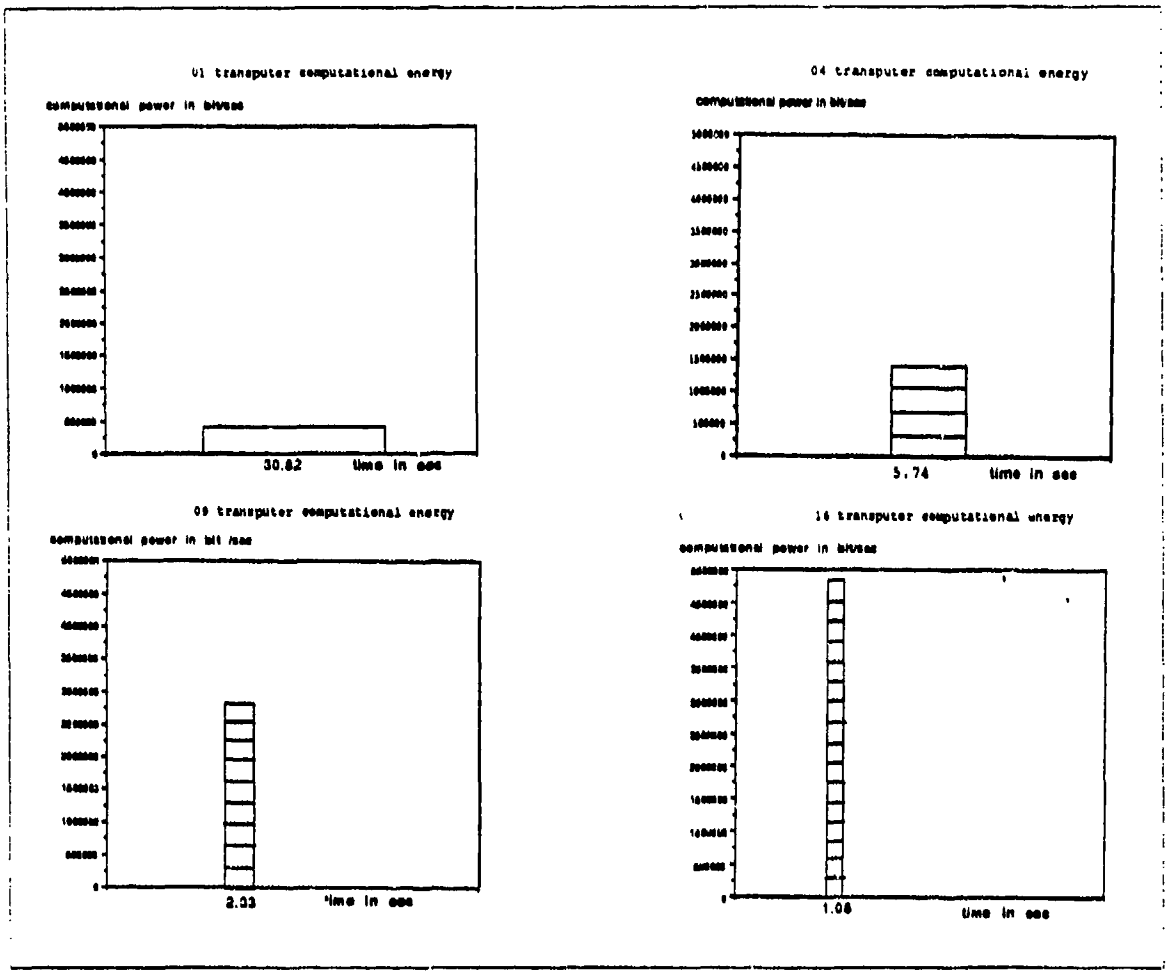


Figure 5.2 Efficiency Comparison

The values for four and nine Transputers are, as explained before, a little bit above the expected because the communication and control overhead, but in the 16 Transputer architecture we see that now the computational power to run the process is a little bit less than the theoretical calculated value, which will be 4968304 bit/sec (310519 x 16).

The reason for that is referred to Chapter IV, on the paragraph "the applications of efficiency analysis", in this case our architecture is entering in the fine granularity zone so the degree at which the parallelism is being exploited is superior to the two former cases; also we can say that for our inductive model, the atomic size of the UCP is for an array of 6 X 6 Transputers, in which the array of temperatures we are deal with is only a 4 X 4 elements. Beyond this point we cannot continue diminishing the size because the Transputer process simply does not work.

From Chapter IV we remember the definition of efficiency; it was the ratio of input computational energy to output computational energy; and also we should realize that the efficiency factor is very low because we have the output information of the process divided by the information delivered by the hardware modules in the time necessary to solve the problem (i.e., time to steady state in our case). In Table 5.2 we can see how the efficiency is improved in relation to the network basis of one Transputer. For this calculation we recall that the input computational energy of the system is equal to the Time times the computational power, and the output computational energy is equal to the maximum data rate for the Transputer which is 1024 x 10^5 bits/sec [INMOSO86], times the Time.

TABLE 5.2

EFFICIENCY COMPARISON FOR THE NETWORKS

in.cp.energy	out.cp.energy	effi. ratio	# Transp.
3155968000	9570195.58	0.0030	01
587776000	8036585.48	0.0137	04
238592000	6592697.72	0.0276	09
110592000	5345404.92	0.0483	16

As can be expected as long as we are entering on the fine granularity zone, the efficiency of the system is improved.

C. RELATIVE EFFICIENCY

Another measure that we performed is the relative efficiency of running our Transputer network procedure in the different systems.

From the definition we know that the relative efficiency of two computer systems is the ratio of the efficiencies of the two systems executing the same process. This results are resume in the Table 5.3, on which we take the higher efficiency as base to compare the others again it.

TABLE 5.3

RELATIVE EFFICIENCY

basis efficiency = 0.0483

relative efficiency for 01 Transp. network = 6.21 %

relative efficiency for 04 Transp. network = 28.36 %

relative efficiency for 09 Transp. network = 57.14 %

On this Table we can realize that the efficiency of the one Transputer network, is about 6.21% the efficiency of the sixteen Transputers network, and so on for the others networks.

The relative efficiency is plot in Figure 5.3. We observe a plot of the efficiencies, related to the highest efficiency presented by the sixteen Transputers network.

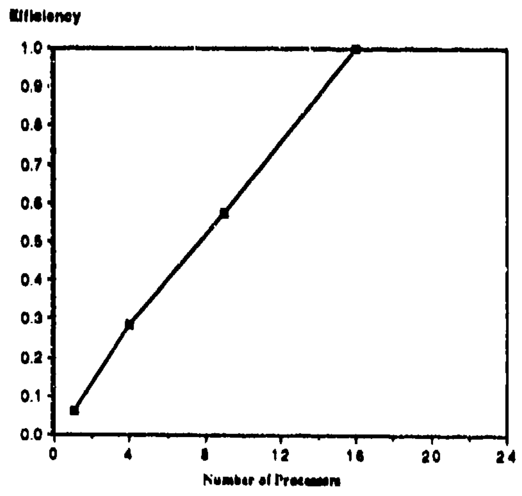


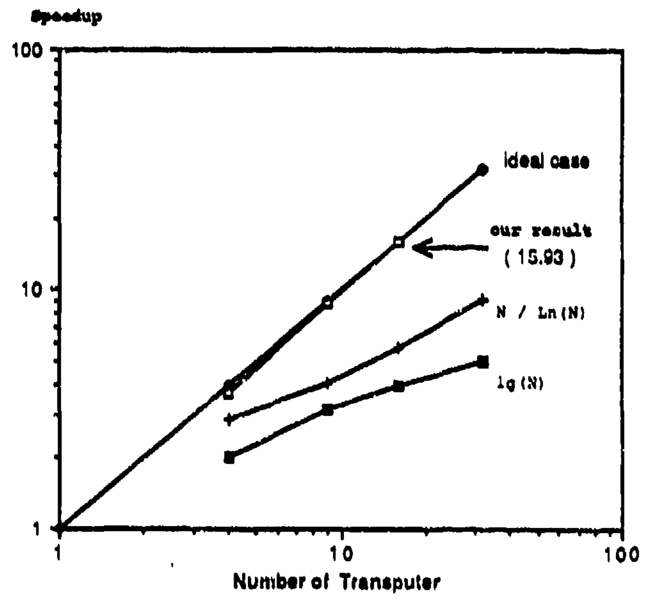
Figure 5.3 Relative Efficiency

D. TRADITIONAL APPROACH TO SPEEDUP ESTIMATION

The speedup that our system is capable of achieving can be graphically determined using the traditional method which is outlined now. We know from before that if we have a parallel computer with N equivalent processors running in parallel on a problem, it will be N times faster than a single processor running the same process. Certainly this is the ideal case, but in the reality the speedup of a system ranges from a lower-bound of lg(N) to an upper-bound of N/ln(N) [KAFA84]. The lower bound is known as Minsky's conjecture. Using this conjecture, we can only expect a speedup of 2 to 4 from our four and sixteen Transputers networks. In the other case we have a better estimate of N/ ln(N). For the latter case let's get through the estimation and subsequently plotting process. We can say that the process at the one Transputer network is running in a unit of time, T1= 1. Let Fi be the probability of assigning the same problem to i processors working equally with an average load di=1/i per processor. Furthermore assume equal probability of each operating mode using i processors, that is Fi= 1/N, for N operating modes : i= 1, 2,..., N. Then the average time required to solve the problem on an N-processor system is given below, where the summation represents N operating modes.

$$T_n = \sum_{i=1}^{n} f_i * d_i = \frac{\sum_{i=1}^{n} \frac{1}{n}}{n}$$

The average speedup S is obtained as the ratio of T1= 1 to Tn; that is S=T1/Tn [KAFA84]. Then in the Figure 5.4 we observe the plot of these upper and lower bound plus the ideal case and also we can see our result.



Time in seconds for each network

seconds	# transputers
1.978×10^{-2}	01
5.336×10^{-3}	04
2.235×10^{-3}	09
1.241×10^{-3}	16

Figure 5.4 Various Estimates of Speedup and our Results

In this plot we can observe, that as we enter in the fine granularity zone, due to the reduction in the communications overhead and computational time, we are exploiting the parallelism in a more efficient fashion, and obtaining a better speedup.

E. SOME DETAILS

There are some conditions about this evaluation and some observations that are necessary to explain and which can serve as hints for future investigation.

First, during the evaluation of the different networks, there were automatic ways of setting up to evaluate. That is the processes were loaded on the Transputer network and when they were ready with the data, they stopped the processes themselves and displayed the information on the screen. Although this look like a fairly good way to save time, in our particular case, the method was discarded because it introduces an overhead in communications which would bias the accuracy of the measurements.

Second, the programs were implemented using the Type INTEGER for all the arithmetic operations. It allows a program to run faster and also the comparison time to establish the "steady state" condition was less than if we had used the Type Floating Point, which from the comparison resulted much more time consuming than the Integer Type, as expected from the OCCAM programming Language specifications.

Third, once the programs were implemented, there were other paths of investigation, such as the one in which the problem size was augmented to run on a 4 X 4 Transputer network, giving an overall array of 96 by 96 elements. In this case the results showed an improvement; i.e., an increasing in throughput was observed.

The reason is simpler but not subtle; in this case the improvement of the performance was due to the fact that the number of computation per unit of time was increasing by a square factor, while the overhead in communications grew in a linear fashion, therefore we were again diminishing the size of the grain. This point will be discussed later in this chapter.

F. COMPARATIVE THROUGHPUT

The throughput is another type of performance measure that can be recorded. The throughput in our system represents the number of results per unit of time that our system can achieve. Table 5.4 gives us a summary of the results.

TABLE 5.4

THROUGHPUT RESULTS

array size	# transp.	throughput	
24 x 24	01	40511	results/sec
12 x 12	04	206580	results/sec
8 x 8	09	392535	results/sec
6 x 6	16	671824	results/sec

Also we can do a relative comparison between the efficiencies as we did before with the efficiencies determined from the computational energy of the system, and certainly, as can be expected, these values are much the same. The summary of this information is recorded on Table 5.5.

TABLE 5.5

THROUGHPUT AND RELATIVE THROUGHPUT

array size	# transp.	rel. throughput
24 x 24	01	6.30 %
12 x 12	04	30.75 %
8 x 8	09	58.43 %
6 x 6	16	---- (**)

(*) the overall array size is the same

(**) basis throughput

G. THE OPTIMAL ZONE

We know that the idea of reducing the granularity in a parallel architecture is the main focus of the research today, but conversely there is a practical limit on how little computational power can be used to execute a process related to the cost of the hardware and the threshold time to execute the process. In other words, it appears to be ideal to break up the problem into smallest possible components for parallel execution, but that fine partitioning can in practice be too costly in terms of overhead and cost of the hardware. For instance, we will be underusing a powerful microprocessor as the Transputer, to solve a very little problem product of this partition, and also when we partitioning a problem very finely, we get more time consumed to communicate data between Transputers, thus slowing down the production of results, and not gaining any improvement in performance. Therefore we have to find a way to balance the communication and computation in a effective manner. To that end, the answer is to get a more relatively coarse partitioning, i.e., get a tradeoff between the maximum number of processors that can be feasibly employed to solve the problem and the time constrains of the problem itself. The idea is to find what we have called the "optimal zone", and operate our machine in it in order to have maximum performance and consequently the best efficiency.

In our sixteen Transputer network prototype, we have a system comprised by many small internal fast memory processing elements or Transputers, that communicate each other relatively fast through the splendid Transputers links, thus this architecture lends itself to fine grained problems. These expectations were confirmed from our experimental results. Another way to approach the problem is to fix the number of processors and reduce the granularity by using a larger array. We use this method in the four network prototype and the sixteen network prototype.

We decreased the granularity using software, i.e., we increased the size of the array of temperatures and we observed and recorded the behavior in relation to throughput.

The testing that was performed on these two architectures was to run the programs, changing granularity starting with a very coarse grain, i.e., we use a temperature array of 4 elements and we incremented its size up to 24 elements, and we were recording and calculating the different throughputs for each different problem size. Thus we could observe the throughput start to increase continuously from the minimum size, and then stabilize at an array size of 14 x 14 elements, for an array of 96 elements (not shown), this behavior still holds. It is true the throughput increased greatly, but on the other hand, the time to solve the problem also increased. Here we have to tie our performance to timing constrains. From this we can deduce the existence of the optimal zone for this type of architecture. To illustrate these concepts we can see in Table 5.6 the results of throughput for different array sizes on both architectures, and in Figure 5.5 we can see a plot of these results.

TABLE 5.6

THROUGHPUT RESULTS FOR DIFFERENT GRAIN SIZE

grain size	array size elements	04 Transp. throughput in results / sec	16 Transp. throughput in results / sec
extrem.large	04	62500	250000
very large	06	111111	444432
large	08	118421	473680
medium	12	135869	543472
transition	14	145161	580640
fine	16	133315	532608
fine	18	140350	561392
very fine	24	144047	576176

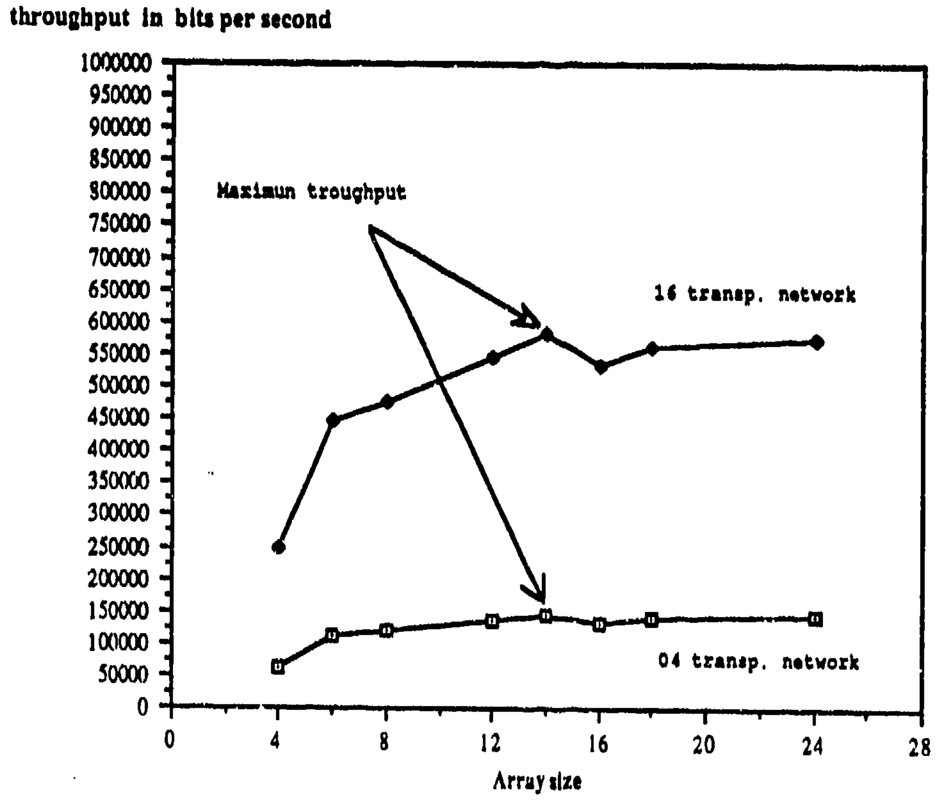


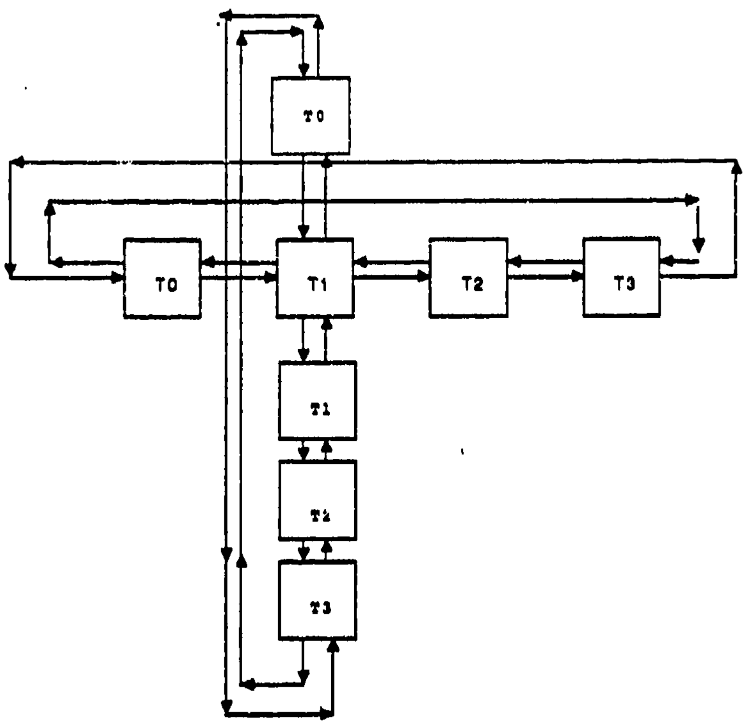
Figure 5.5 Throughput for Different Grain Size in a 04 and a 16 Transputer Network

H. HOW THE PARALLELISM WAS ACCOMPLISHED

So far, we discussed throughput and speedup of the different Transputer networks and we have proved from the experimental results the existence of parallel activity. Now let's consider the parallelism in more detail to have a clear idea of what is going on.

We must recall that we have a Transputer network process running making use of the virtual shared memory system. This virtual shared memory is obtained by duplicating information in local memories so that when a producing process writes a new value into its local memory, the synchronous operating system generates a message which is broadcast to all consumers of the data via the point to point link mechanism of the Transputers. Thus the local memory of each computing node (Transputer in the network) contains the duplicate data ready to be consumed by each consumer in its local memory, [KOD88]. The reading and writing is accomplished in every complete cycle of communication and calculation, and is executed in a carefully synchronized fashion so that the writing of the data structure by a producer is completed before that data structure is read by the consumer [REKA79]. In our heat flow problem this sequence of events occurs in the following way: suppose we map an imaginary grid over the plate denoting at each line intersection a Transputer which is in charge of calculating a square segment of temperatures for the plate.

As soon as some process is ready with the updating of its set of temperatures due to a previous boundary exchange with its neighbors, it proceeds to calculate the new temperatures, updating its internal array of temperatures (updating its local memory, represented by the data structure which contains the array of temperatures). It is then ready for a new cycle, which always starts with the boundaries exchange (write in and read from the local memory of its neighbors). This last action cannot be seen as a local activity which only affects the state of the neighbors of this process but as a kind of chain reaction which is propagated in vertical and horizontal sense all over the network, creating the so called virtual shared memory effect. We can observe that assertion in Figure 5.6 on next page.



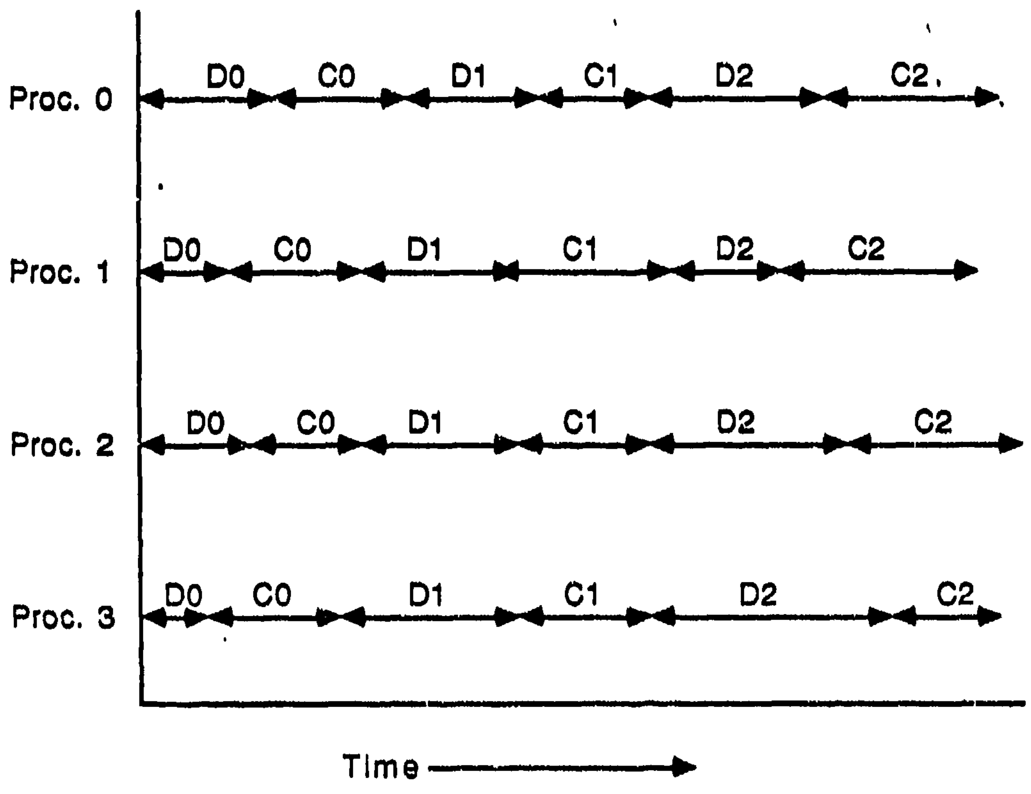
For simplicity only a row of the network is displayed, but this transfer of boundaries, which in turn represent the update mechanism, must be observed as a simultaneous process in both directions, (up--down and right-- left).

Figure 5.6 Memory Updating Mechanism in the Network

Let's describe what we mean with chain reaction in a more precise way: suppose at some instant of time the process 0 receives and sends (writes in i.s local memory and writes out the surrounding local memories) the boundaries from/to its neighbors. The following processor (or immediate neighbor on bottom or right) let's call and locate it to the right, process 01 which does exactly the same to its right and the next process receive and send these boundaries behaving in the same way until we arrive at the end of the row, which we call the end for the sake of the illustration of the concept. In reality if we look in more detail, we shall agree that this end of the row does not exist, because the last Transputer is physically connected to the first Transputer in a closed loop. Moreover, this movement of data to the right is also registered in the opposite sense concurrently (from these notions were established the name of "double transitive closure"). Thus we can assert that at any instant of time each Transputer in the network updates or writes into the local memories of the other Transputers in the network due to a kind of interactive total exchange of boundaries. In other words, when Transputer 0 receives the boundaries from Transputer 01 at its right, it is receiving not only the effect of the boundary temperatures of this Transputer but also the effect of boundary temperatures in Transputer 02, and Transputer 03, and so forth in a concurrent fashion, yielding a kind of instantaneous daisy chain transmission.

We can assert that the time in which memory was last updated in process 0 due to the data produced in process 3 at the end of the row is the very same as the time for updating of memory in process 3 due to the data produced by process 0. In the timing diagram of Figure 5.7 is shown the concurrent activity of the sixteen Transputer network prototype, using for the sake of simplicity only a row of the array. It must be remembered that the activity occurs concurrently in a vertical and horizontal sense, in right to left and top to bottom directions, and vice versa.

The symbol C stands for calculations and the symbol D for updated data value. During the first complete cycle, process 0 updates its data value, receiving information via link2 from process 01 and process 01 at the same time receives this information for its own consumption from Transputer 0 via link3. This activity is performed concurrently. At the same time, process 01 does the same for process 02, and process 02 for process 03. After that, we observe a parallel calculation activity in the fourth process, which will last, at a maximum, the time which takes the last process to achieve its calculations. This does not means that the next iteration will be delayed by any processor calculation other than the process 0 calculation, which is in charge to start the cycle. Therefore the calculation activity of the slower process may overlap in time with the updating data time of the other processes for the next updating cycle, and the other way around. That is, the updating activity of a process may overlap with the calculating activity of the other process, but the bottom line for this overlapping is that it is not possible to perform an updating activity which belongs to a determined cycle with the calculations data of the same cycle.



The Parallelism is easily observed by the overlapping in time of the update and computational periods of the different processes.

Figure 5.7 Timing Diagram

I. THE TIMING CONSTRUCT

To finish, we wish to describe the timing construct used to obtain the measurements.

The T414 Transputer has two timers; a high priority timer with a resolution of one microsecond and a cycle time of about 71 minutes, and a low priority timer which has a resolution of 64 x 10^ -6 seconds and a cycle time of 37 hours. The timer used was the low priority timer, and the type of construct was an elapsed time construct to determine the elapsed time from start to finish of some activities within the process. The basic structure of this construct can be seen in Figure 5.8.

```
.... Declaration of Variables
 Timer  clock:
 INT time1,time2,timetest:
 SEQ
   clock ? time1
   .... timing code
   clock ? time2
   ..... more code
   timetest := time2 - time1     (final result)
```

Figure 5.8 The Timing Construct

Essentially the timing construct has two variables of integer type, (time2, time1) which are used to store the value of the Timer and a third integer variable called timetest which give us the difference , which is the value of interest.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The conclusions that we obtain from our observations during this research were as follow:

First, the effects of parallelism in the networks were proved practically and theoterically.

Second, the existence of an optimal zone related to the granularity of the system and time constrains of the problem was predicted in theory and deduced from the experimental results.

Third, the degree of parallelism attained in these networks is quite remarkable, as shown in the Figures due to speedup and efficiency. For example, in the 16 Transputer network prototype we obtain for a 6 by 6 array of temperatures a throughput of 671824 results per second. Considering the fact that we perform 7 arithmetic operations per result, (5 additions, one division and one multiplication), that gives us 4,702,768 integer operations per second. Also should be taken under consideration that because the fact of the implementation "march forward in time", was necessary to copy the entire array of temperatures into a temporary array which is later transferred to the real array of temperatures, thus that represents an overhead which slows down the process significantly.

Fourth, the improvement in performance is a trade off between the number of processors (Transputer) added to the network and the granularity on one hand, and on the other hand, the cost of the hardware and the time constrains of the problem.

Fifth The Transputer network is an architecture comprised of many small internal fast memory processing elements that communicate to each other through the powerful Transputer links. Thus this architecture lends itself to fine grained problems.

B. POSSIBILITIES OF THE TRANSPUTER

At the beginning of this thesis some guidelines about the importance of the Transputer were given.

The real importance of the Transputer lies in the fact that it represents a new level of abstraction in the physical design of information systems. As we know so far, there have been two levels of abstraction:

1) the electronic component, in which the information is represented in terms of electrical signals, like voltage or capacitance, and

2) the logical gate, in which the information is represented by logical levels, so the electrical details have been abstracted from the design process.

The Transputer offers a third level of abstraction, based on language, where the basis unit is the word, which can be given specific semantic connotations by the provision of an appropriate set of information operations. Therefore the Transputer chip will be used as time goes on in much the same way as the discrete transistor was used about 20 years ago.

C. RECOMMENDATIONS

Bear in mind that the fundamental research reason of the AEGIS modeling group at the NPS, is to develop a suitable replacement the older architectures on board the Ticonderoga class ships. It is recommended that rather than broadening the Transputer Laboratory to cope with this function, the research should be divided into specific smaller projects which help to implement the new system. This recommendation is basically due to the limited availability of resource for a small group like this.

Another recommendation is to seek for feasible research projects related to weapons that can be developed by the Group.

It is also important to continue the trend of this thesis in following the exploration of this type of architecture and the production of software for it.

It will be interesting to see how this type of architecture can handle problems as weather forecasting for a particular weather model. Finally is important to continue research in the field of graphic applications, especially that which pertains to the study of Chaotic Systems such as Mandelbrot and Julia sets.

APPENDIX A

01 TRANSPUTER NETWORK SOURCE CODE

```
---------------------------------------------------------------
  PROC input.handler (CHAN OF ANY keyboard,screen)
---------------------------------------------------------------

    #USE  "c:\tdsiolib\userio.tsr":
    VAL link0out  IS 0:
    VAL link1out  IS 1:
    VAL link2out  IS 2:
    VAL link3out  IS 3:
    VAL link0in   IS 4:
    VAL link1in   IS 5:
    VAL link2in   IS 6:
    VAL link3in   IS 7:
    CHAN OF ANY leftin,rightout,antirightout,antileftin:
    PLACE leftin AT link3in:
    PLACE rightout AT link3out:
    PLACE antirightout AT link2out:
    PLACE antileftin AT link2in:
    BOOL turning:
    VAL s IS 11:
    VAL esc IS 223:
    VAL g IS 333:
    VAL size IS 24:
    INT w,tag,he,no,z,txt:
    [size] INT temp:
    [size] INT recp:
    [size] INT recp1:
    [size] INT recp2:
    [size] [size] INT truly0:
    SEQ
      no:=0
      write.full.string (screen, " Enter the hot end
                                temperature ")
      read.echo.int (keyboard,screen,he,no)
      newline(screen)
      no:=0
      write.full.string (screen, " Enter the propagation
                               rate ")
      read.echo.int (keyboard,screen,w,no)
      newline(screen)
      SEQ
        SEQ r = 0 FOR size
          SEQ
            temp [r] := 0
            recp [r] := 0
            recp1 [r] := 0
            recp2 [r] := 0
    SEQ r = 0 FOR size
      temp [r] := he
    tag:= g
    antirightout ! tag;w;temp
    rightout ! tag;w;temp
    antirightout ! recp2
    rightout ! recp1
    turning := TRUE
    SEQ
      WHILE turning
        PRI ALT
          keyboard ? z
            SEQ
              IF
                z = esc
                  SEQ
                    tag:= s
                    antileftin ? recp
                    leftin ? recp
                    antirightout ! tag;w;temp
                    rightout ! tag;w;temp
                    antileftin ? truly0
                    SEQ r = 0 FOR size
                      SEQ
                        SEQ c = 0 FOR size
                          SEQ
                            txt:= truly0 [r] [c]
                            write.int (screen,txt,4)
                        newline(screen)
                    turning := FALSE
                    newline(screen)
                TRUE
                  SKIP
          antileftin ? recp1
            SEQ
              leftin ? recp2
              antirightout ! tag;w;temp
              rightout ! tag;w;temp
              antirightout ! recp2
              rightout ! recp1
  newline(screen)
  write.full.string(screen, "Type ANY to return to TDS")
  INT any:
  read.char(keyboard, any)
:
```

```
VAL link0out  IS 0:
VAL link1out  IS 1:
VAL link2out  IS 2:
VAL link3out  IS 3:
VAL link0in   IS 4:
VAL link1in   IS 5:
VAL link2in   IS 6:
VAL link3in   IS 7:
[9] CHAN OF ANY channel,antichannel:
--------------------------------------------------------
PROC central.node(VAL  INT engine,CHAN OF ANY
                       leftin,topin,rightin,bottomin,
                  leftout,topout,rightout,bottomout)
----------------------------------------------------------
  #USE  "c:\tdsiolib\userio.tsr":
  BOOL active :                     --Declarations
  VAL s IS 11:
  VAL g IS 333:
  VAL size IS 24:
  INT tag,w,tp,n:
  [size] [size] INT square:
  [size] [size] INT calcul:
  [size] INT dummy0:
  [size] INT dummy1:
  [size] INT sender0:
  [size] INT sender1:
  [size] INT sender2:
  [size] INT sender3:
  WHILE TRUE
    SEQ          -- Array initialization
      SEQ r= 0 FOR size
        SEQ c= 0 FOR size
          SEQ
            square [r] [c] := 0
            calcul [r] [c] := 0
      SEQ r= 0 FOR size
        SEQ
          dummy0 [r] := 0
          dummy1 [r] := 0
          sender0 [r] := 0
          sender1 [r] := 0
          sender2 [r] := 0
          sender3 [r] := 0
      active:= TRUE
      n:= engine
```

```
WHILE active
  SEQ
    IF
      n= 0
        SEQ
          topin ? tag;w;dummy1
          rightout ! tag
          IF
            tag= s
              SEQ
                active:= FALSE
                topout ! square
            TRUE       -- Communication block
              SEQ
                PAR
                  leftin ?    dummy0
                  topin ?     dummy0
                  rightin ?   dummy0
                  bottomin ?  dummy0
                  leftout !   sender0
                  topout !    sender1
                  rightout !  sender2
                  bottomout ! sender3
                SEQ r = 0 FOR size
                  SEQ
                    square[r] [0] := dummy1 [r]
                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
                    SEQ
                      tp:= ((w * square [r] [c]   ) +
                        ( square [r] [c-1] +( square
                         [r] [c + 1] square[r-1][c]
                             + square [r + 1] [c]
                       )))) /  (4 + w)
                      calcul [r] [c] := tp
                SEQ r = 0 FOR size
                  calcul [r] [0]:= square[r] [0]
                square := calcul
                SEQ r = 0 FOR size
                  SEQ
                    sender0 [r]:= square[r] [1]
                    sender1 [r]:= square[1] [r]
                    sender2 [r]:=  square[r] [size -
                                  2]
                   sender3 [r]:=square[size - 2] [r]
:
```

```
--------------------------------------------------------
PROC transp.horizontal (VAL INT engine, CHAN OF ANY
                 leftin,topin,rightin,bottomin,leftout,
                        topout,rightout,bottomout)
--------------------------------------------------------

  #USE  "c:\tdsiolib\userio.tsr":
  BOOL active:
  VAL s IS 11:
  VAL g IS 333:
  VAL size IS 24:
  INT tag,w,n:
  [size] INT spec1:
  [size] INT spec2:

  WHILE TRUE
    SEQ
      SEQ r = 0 FOR size
        SEQ
          spec1 [r] := 0
          spec2 [r] := 0
      n:= engine
      active:= TRUE
      tag:= g
      WHILE active
        SEQ
          IF
            n= 2
              SEQ
                leftin ? tag
                IF
                  tag= s
                    active:= FALSE
                  TRUE
                    SEQ
                      leftin ? spec1
                      rightin? spec2
                      leftout ! spec2
                      rightout ! spec1
            n= 3
              SEQ
                leftin ? tag
                IF
                  tag = s
                    active:= FALSE
                  TRUE
                    SKIP
:
```

```
-------------------------------------------------------------
PROC transp.vertical   (VAL INT engine, CHAN OF ANY
                        leftin,topin,rightin,bottomin,
                        leftout,topout,rightout,
                        bottomout)
-------------------------------------------------------------
#USE  "c:\tdsiolib\userio.tsr":
  BOOL active:                              -- Variable declaration
  VAL s IS 11:
  VAL g IS 333:
  VAL size IS 24:
  INT tag,w,n:
  [size] INT spec1:
  [size] INT spec2:
  WHILE TRUE
    SEQ
      SEQ r = 0 FOR size
        SEQ
          spec1 [r] := 0
          spec2 [r] := 0
      n:= engine
      active:= TRUE
      tag:= g
      WHILE active
        SEQ
          IF
            n= 1
              SEQ
                bottomin ? tag;w;spec1
                rightout ! tag
                IF
                  tag= s
                    active:= FALSE
                  TRUE
                    SEQ
                      bottomin ? spec1
                      topout ! spec1
                      topin ? spec2
                      bottomout ! spec2
:
 ------------------------------------------------------------
     -- Processor Placement
  -----------------------------------------------------------
PLACED PAR
    PROCESSOR 0 T4
      PLACE channel[0] AT link0in:
      PLACE channel[1] AT link1in:
      PLACE channel[2] AT link2in:
      PLACE channel[3] AT link3in:
      PLACE antichannel[0] AT link0out:
      PLACE antichannel[1] AT link1out:
      PLACE antichannel[2] AT link2out:
      PLACE antichannel[3] AT link3out:
central.node(0,channel[0],channel[1],channel[2],channel[3],
                 antichannel[0],antichannel[1],
                 antichannel[2],antichannel[3])

    PROCESSOR 1 T4
      PLACE channel[4] AT link0in:
      PLACE channel[5] AT link1in:
      PLACE channel[3] AT link2out:
      PLACE channel[6] AT link3in:
      PLACE antichannel[4] AT link0out:
      PLACE antichannel[5] AT link1out:
      PLACE antichannel[3] AT link2in:
      PLACE antichannel[6] AT link3out:
transp.vertical(1,channel[5],antichannel[3],channel[6],
                      channel[4],antichannel[5],channel[3],
                      antichannel[6],antichannel[4])

    PROCESSOR 2 T4
      PLACE channel[7] AT link0in:
      PLACE channel[0] AT link1out:
      PLACE channel[8] AT link2in:
      PLACE channel[2] AT link3out:
      PLACE antichannel[7] AT link0out:
      PLACE antichannel[0] AT link1in:
      PLACE antichannel[8] AT link2out:
      PLACE antichannel[2] AT link3in:
      transp.horizontal(2,antichannel[2],channel[7],
                          antichannel[0],channel[8],
                          channel[2],antichannel[7],
                          channel[0],antichannel[8])
```

```
PROCESSOR 3 T4
  PLACE channel[5] AT link0out:
  PLACE channel[7] AT link1out:
  PLACE channel[6] AT link2out:
  PLACE channel[8] AT link3out:
  PLACE antichannel[5] AT link0in:
  PLACE antichannel[7] AT link1in:
  PLACE antichannel[6] AT link2in:
  PLACE antichannel[8] AT link3in:
  transp.horizontal(3,antichannel[6],antichannel[8],
  antichannel[5],antichannel[7],channel[6],
  channel[8],channel[5],channel[7])
```

APPENDIX B

04 TRANSPUTER NETWORK SOURCE CODE

```
-----------------------------------------------------------------
  PROC input.handler (CHAN OF ANY keyboard,screen)

This procedure handles the input and output from the
Transputer
network.
-----------------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    VAL link0out  IS 0:                    --- Variable
    VAL link1out  IS 1:                --- Declarations
    VAL link2out  IS 2:
    VAL link3out  IS 3:
    VAL link0in   IS 4:
    VAL link1in   IS 5:
    VAL link2in   IS 6:
    VAL link3in   IS 7:
    CHAN OF ANY leftin,rightout,antirightout,antileftin:
    PLACE leftin AT link3in:
    PLACE rightout AT link3out:
    PLACE antirightout AT link2out:
    PLACE antileftin AT link2in:
    BOOL go,turning:
    VAL s IS 11:
    VAL esc IS 223:
    VAL g IS 333:
    VAL size IS 12:
    INT w,tag,he,no,z,counter,txt:
    [size] INT temp:
    [size] INT recp:
    [size] INT recp1:
    [size] INT recp2:
    [size] [size] INT truly0:
    [size] [size] INT truly1:
    [size] [size] INT truly2:
    [size] [size] INT truly3:
    [size] [size] INT tx:
    SEQ
      no:=0
      write.full.string (screen,  " Enter the hot end
                                 temperature")
      read.echo.int (keyboard,screen,he,no)
      newline(screen)
      no:=0
      write.full.string (screen, " Enter the propagation
                            rate ")
      read.echo.int (keyboard,screen,w,no)
      newline(screen)
```

```
        SEQ
          SEQ r = 0 FOR size              Array initialization
            SEQ
              temp [r] := 0
              recp [r] := 0
              recp1 [r] := 0
              recp2 [r] := 0
          SEQ r = 0 FOR size
            temp [r] := he
          tag:= g
          antirightout ! tag;w;temp
          rightout ! tag;w;temp
          antirightout ! recp2
          rightout ! recp1
          turning := TRUE
          SEQ
            WHILE turning
              PRI ALT
                keyboard ? z
                  SEQ
                    IF
                      z = esc
                        SEQ
                          SEQ
                            tag:= s
                            antileftin ? recp
                            leftin ? recp
                            antirightout ! tag;w;temp
                            rightout ! tag;w;temp
                            counter := 0
                            leftin ? truly0;truly2;
                                     truly1;truly3
                            WHILE counter < 4
                              SEQ
                                SEQ r = 0 FOR size - 1
                                  SEQ
                                    SEQ c = 0 FOR size - 1
----   Printing the temp.                 SEQ
----   array                              tx:= truly0
                                        txt:= tx [r] [c]
                                        write.int (screen,
                                                     txt,5)
                                        SEQ l = 1 FOR size - 1
                                      SEQ
                                        tx:= truly2
                                        txt:= tx [r] [l]
                                        write.int (screen,
                                                   txt,5)
                                      newline(screen)
```

```
                           SEQ r = 1 FOR size - 1
                             SEQ
                               SEQ d = 0 FOR size - 1
                                 SEQ
                                   tx:= truly1
                                   txt:= tx [r] [d]
                                   write.int (screen,
                                              txt,5)
                                   SEQ h = 1 FOR size - 1
                                 SEQ
                                   tx:= truly3
                                   txt:= tx [r] [h]
                                   write.int (screen,
                                              txt,5)
                                newline(screen)
                           counter:= counter + 4
                       turning := FALSE
                     newline(screen)
           antileftin ? recp1
             SEQ
               leftin ? recp2
               antirightout ! tag;w;temp
               rightout ! tag;w;temp
               antirightout ! recp2
               rightout ! recp1
   newline(screen)
   write.full.string(screen, "Type ANY to return to TDS")
  INT any:
   read.char(keyboard, any)
:
```

```
------------------------------------------------------------
  VAL link0out  IS 0:
  VAL link1out  IS 1:
  VAL link2out  IS 2:              --- Channel declaration
  VAL link3out  IS 3:
  VAL link0in   IS 4:
  VAL link1in   IS 5:
  VAL link2in   IS 6:
  VAL link3in   IS 7:
  [9] CHAN OF ANY channel,antichannel:

------------------------------------------------------------

  PROC   central.node(VAL    INT    engine,    CHAN    OF    ANY
leftin,topin,

rightin,bottomin,leftout,topout,rightout,
                    bottomout)
------------------------------------------------------------

    #USE  "c:\tdsiolib\userio.tsr":
    BOOL active :
    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 12:
    INT tag,w,tp,n:
    [size] [size] INT square:
    [size] [size] INT calcul:
    [size] INT dummy0:
    [size] INT dummy1:
    [size] INT dummy2:
    [size] INT dummy3:
    [size] INT dummy4:
    [size] INT sender0:
    [size] INT sender1:
    [size] INT sender2:
    [size] INT sender3:
    [size] [size] INT temporal:
    WHILE TRUE
      SEQ
        SEQ r= 0 FOR size
          SEQ c= 0 FOR size
            SEQ
              square [r] [c] := 0
              calcul [r] [c] := 0
              temporal [r] [c] := 0
        SEQ r= 0 FOR size
          SEQ                              --- Array
            dummy0 [r] := 0        -- Initialization
            dummy1 [r] := 0
            dummy2 [r] := 0
            dummy3 [r] := 0
        dummy4 [r] := 0
        sender0 [r] := 0
        sender1 [r] := 0
        sender2 [r] := 0
        sender3 [r] := 0
    active:= TRUE
    n:= engine
    WHILE active
      SEQ
        IF
          n= 0
            SEQ
              topin ? tag;w;dummy1
              rightout ! tag;w
              IF
                tag= s
                  active:= FALSE
                TRUE
                  SEQ
                    PAR
                      leftin ?    dummy0 --Communication
                      topin ?     dummy4   -- Block
                      rightin ?   dummy2
                      bottomin ?  dummy3
                      leftout !   sender0
                      topout !    sender1
                      rightout !  sender2
                      bottomout ! sender3
                    SEQ r = 0 FOR size
                      SEQ
                        square[r] [0] := dummy1 [r]
                        square[r] [size - 1]  := dummy2
                                                  [r]
                         square[size -  1] [r] := dummy3
                                                    [r]
                     SEQ r = 1 FOR size - 2
                                           -- Calculations
                SEQ c = 1 FOR size - 2
                  SEQ
                    tp:= ((w * square [r] [c] ) + (
                               square[r][c-1] +  square
                           [r] [c + 1] + ( square [r-1]
                           [c]  + square [r + 1] [c]
                           )))) / (4 + w)
                    calcul [r] [c] := tp
                    SEQ r = 0 FOR size
                      calcul [r] [0]:= square[r] [0]
                    square := calcul
                    SEQ r = 0 FOR size
                      SEQ
                        sender0 [r]:= square[r] [1]
                        sender1 [r]:= square[1] [r]
                    sender2  [r]:= square[r][size-2]
                    sender3  [r]:= square[size-2][r]
  n= 1
    SEQ
      bottomin ? tag;w;dummy3
      rightout ! tag;w
      IF
        tag= s
          active:= FALSE
        TRUE
          SEQ
            PAR
              leftin ?    dummy0
              topin ?     dummy1
              rightin ?   dummy2
              bottomin ?  dummy4
              leftout !   sender0
              topout !    sender1
              rightout !  sender2
              bottomout ! sender3
            SEQ r = 0 FOR size
              SEQ
                square[r] [0] := dummy3 [r]
                square[0] [r] := dummy1 [r]
                square[r] [size-1] := dummy2 [r]
            SEQ r = 1 FOR size - 2
              SEQ c = 1 FOR size - 2
                SEQ
                  tp:= ((w * square[r][c])
                       +(square[r] [c-1] +
                       ( square[r][c+1] +
                       (square[r-1][c] +
                       square [r + 1] [c] )))) /
                       (4+w)
                  calcul [r] [c] := tp
            SEQ r = 0 FOR size
              calcul [r] [0] := square [r]  [0]
          square := calcul
            SEQ r = 0 FOR size
              SEQ
                sender0 [r] := square [r] [1]
                sender1 [r] := square [1] [r]
                sender2 [r] := square[r][size-
                                             2]
                sender3 [r] := square [size - 2]
                                               [r]
```

```
n= 2
    SEQ
      leftin ? tag;w
      IF
        tag= s
          active:= FALSE
        TRUE
          SEQ
            PAR
              leftin ?    dummy0
              topin ?     dummy1
              rightin ?   dummy2
              bottomin ?  dummy3
              leftout !   sender0
              topout !    sender1
              rightout !  sender2
              bottomout ! sender3
            SEQ r = 0 FOR size
              SEQ
                square[r] [0] := dummy0 [r]
                square[size -1][r] := dummy3 [r]
              SEQ r = 1 FOR size - 2
              SEQ c = 1 FOR size - 2
                SEQ
                  tp:= ((w * square[r][c])  + (
                      square  [r] [c-1]+( square
                         [r] [c + 1]  + (square
                    [r-1][c] + square[r + 1]
                      [c] )))) / (4 + w)
                  calcul [r] [c] := tp
            square := calcul
            SEQ  r = 0 FOR size
              SEQ
                sender0 [r] := square [r] [1]
                sender1 [r] := square [1] [r]
                sender2 [r] := square[r][size-2]
                sender3 [r] := square [size - 2]
                                                [r]
```

```
n= 3
  SEQ
    leftin ? tag;w
    IF
      tag= s
        active:= FALSE
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender1
            rightout !  sender2
            bottomout ! sender3
          SEQ r = 0 FOR size
            SEQ
              square[0] [r] := dummy1 [r]
              square[r] [0] := dummy0 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square[r] [c] ) + (
                    square [r][c-1] + ( square
                     [r] [c + 1]+(square [r-1]
                     [c] +  square [r + 1] [c]
                    ))))/(4 + w)
                 calcul [r] [c] := tp
          square := calcul
          SEQ  r = 0 FOR size
            SEQ
              sender0 [r] := square [r] [1]
              sender1 [r] := square [1] [r]
              sender2 [r] := square[r][size-2]
              sender3 [r] := square [size - 2]
                                             [r]
          IF
            n=0
              SEQ
                bottomout ! square
                rightin ? temporal
                bottomout ! temporal
            n=2
              SEQ
                leftout ! square
            n=3
              SEQ
                rightout ! square
            n=1
              SEQ
                topin ? temporal
                bottomout ! temporal
                topin ? temporal
                bottomout ! temporal
                bottomout ! square
                leftin ? temporal
                bottomout ! temporal
  :
```

```
------------------------------------------------------------
     -- Processors Placement
------------------------------------------------------------

  PLACED PAR
    PROCESSOR 0 T4
      PLACE channel[0] AT link0in:
      PLACE channel[1] AT link1in:
      PLACE channel[2] AT link2in:
      PLACE channel[3] AT link3in:
      PLACE antichannel[0] AT link0out:
      PLACE antichannel[1] AT link1out:
      PLACE antichannel[2] AT link2out:
      PLACE antichannel[3] AT link3out:
      central.node(0,channel[0],channel[1],channel[2],
                   channel[3],antichannel[0],antichannel[1],
                   antichannel[2],antichannel[3])
```

```
PROCESSOR 1 T4
  PLACE channel[4] AT link0in:
  PLACE channel[5] AT link1in:
  PLACE channel[3] AT link2out:
  PLACE channel[6] AT link3in:
  PLACE antichannel[4] AT link0out:
  PLACE antichannel[5] AT link1out:
  PLACE antichannel[3] AT link2in:
  PLACE antichannel[6] AT link3out:
  central.node(1,channel[5],antichannel[3],
               channel[6],channel[4],antichannel[5],
               channel[3],antichannel[6],antichannel[4])


PROCESSOR 2 T4
  PLACE channel[7] AT link0in:
  PLACE channel[0] AT link1out:
  PLACE channel[8] AT link2in:
  PLACE channel[2] AT link3out:
  PLACE antichannel[7] AT link0out:
  PLACE antichannel[0] AT link1in:
  PLACE antichannel[8] AT link2out:
  PLACE antichannel[2] AT link3in:
  central.node(2,antichannel[2],channel[7],
               antichannel[0],channel[8],channel[2],
               antichannel[7],channel[0],antichannel[8])


PROCESSOR 3 T4
  PLACE channel[5] AT link0out:
  PLACE channel[7] AT link1out:
  PLACE channel[6] AT link2out:
  PLACE channel[8] AT link3out:
  PLACE antichannel[5] AT link0in:
  PLACE antichannel[7] AT link1in:
  PLACE antichannel[6] AT link2in:
  PLACE antichannel[8] AT link3in:
  central.node(3,antichannel[6],antichannel[8],
               antichannel[5],antichannel[7],channel[6],
               channel[8],channel[5],channel[7])
```

APPENDIX C

09 TRANSPUTER NETWORK SOURCE CODE

```
------------------------------------------------------------
  PROC input.handler (CHAN OF ANY keyboard,screen)
------------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    VAL link0out  IS 0:
    VAL link1out  IS 1:
    VAL link2out  IS 2:
    VAL link3out  IS 3:
    VAL link0in   IS 4:
    VAL link1in   IS 5:
    VAL link2in   IS 6:
    VAL link3in   IS 7:
    CHAN OF ANY leftin,rightout,antirightout,antileftin:
    PLACE leftin AT link3in:
    PLACE rightout AT link3out:
    PLACE antirightout AT link2out:
    PLACE antileftin AT link2in:
    BOOL go,turning:
    VAL s IS 11:
    VAL esc IS 223:
    VAL g IS 333:
    VAL size IS 8:
    INT w,tag,he,no,z,counter,counter1,txt:
    [size] INT temp:
    [size] INT recp:
    [size] INT recp1:
    [size] INT recp2:
    [size] [size] INT truly:
    [9][size] [size] INT true:
    SEQ
      no:=0
      write.full.string (screen, " Enter the hot end
                                temperature ")
      read.echo.int (keyboard,screen,he,no)
      newline(screen)
      no:=0
      write.full.string (screen, " Enter the propagation
                                rate ")
      read.echo.int (keyboard,screen,w,no)
      newline(screen)
      SEQ                          -- Array initialization
        SEQ r = 0 FOR size
          SEQ
            temp [r] := 0
            recp [r] := 0
            recp1 [r] := 0
            recp2 [r] := 0
        SEQ r = 0 FOR size
  temp [r] := he
tag:= g
antirightout ! tag;w;temp
rightout ! tag;w;temp
antirightout ! recp2
rightout ! recp1
turning := TRUE
SEQ
  WHILE turning
    PRI ALT
      keyboard ? z
        SEQ
          IF
            z = esc
              SEQ
                SEQ
                  tag:= s
                  antileftin ? recp
                  leftin ? recp
                  antirightout ! tag;w;temp
                  rightout ! tag;w;temp
                  counter := 0
                  counter1 := 0
                  WHILE counter < 9
                    SEQ
                      antileftin ? truly
                      SEQ h = 0 FOR size
                        SEQ p = 0 FOR size
                          true [counter] [h] [p] :=
                                      truly [h][p]
                      counter := counter + 1
                  SEQ
                    SEQ r = 0 FOR size - 1
                      SEQ
                        SEQ c = 0 FOR size -
                          SEQ
                            txt:= true[counter1] [r]
                                                 [c]
                            write.int (screen,txt,3)
                        SEQ l = 1 FOR size - 2
                          SEQ
                            txt:= true  [counter1 +
                                       3] [r][l]
                            write.int (screen,txt,3)
                        SEQ d = 1 FOR size - 1
                          SEQ
                            txt:= true  [counter1 +
                                       6] [r] [d]
                            write.int (screen,txt,3)
                        newline(screen)
                      counter1:= counter1 + 1
            SEQ r = 1 FOR size - 2
```

```
                              SEQ
                                SEQ c := 0 FOR size - 1
                                  SEQ
                                    txt:= true [counter1]
                                                  [r] [c]
                                    write.int (screen,txt,3)
                                SEQ l = 1 FOR size - 2
                                  SEQ
                                    txt:= true  [counter1 +
                                              3] [r] [l]
                                    write.int (screen,txt,3)
                                SEQ d = 1 FOR size - 1
                                  SEQ
                                    txt:= true [counter1 +
                                                 6] [r][d]
                                    write.int (screen,txt,3)
                                newline(screen)
                            counter1:= counter1 + 1
                            SEQ r = 1 FOR size - 1
                              SEQ
                                SEQ c = 0 FOR size - 1
                                  SEQ
                                    txt:= true [counter1]
                                                  [r] [c]
                                    write.int (screen,txt,3)
                                SEQ l = 1 FOR size - 2
                                  SEQ
                                    txt:= true [counter1 +
                                                3] [r][l]
                                    write.int (screen,txt,3)
                                SEQ d = 1 FOR size - 1
                                  SEQ
                                    txt:= true [counter1 +
                                                 6] [r] [d]
                                    write.int (screen,txt,3)
                                newline(screen)
                            counter1:= counter1 + 1
                          turning := FALSE
                        newline(screen)
               antileftin ? recp1
                 SEQ
                   leftin ? recp2
                   antirightout ! tag;w;temp
                   rightout ! tag;w;temp
                   antirightout ! recp2
                   rightout ! recp1
      newline(screen)
      write.full.string(screen, "Type ANY to return to TDS")
      INT any:
      read.char(keyboard, any)
  :
```

```
-----------------------------------------------------------------
  VAL link0out  IS 0:
  VAL link1out  IS 1:
  VAL link2out  IS 2:     -- Channel declaration
  VAL link3out  IS 3:
  VAL link0in   IS 4:
  VAL link1in   IS 5:
  VAL link2in   IS 6:
  VAL link3in   IS 7:
  [35] CHAN OF ANY channel,antichannel:
-----------------------------------------------------------------
-----------------------------------------------------------------
  PROC   central.node(VAL   INT   engine,   CHAN   OF   ANY
leftin,topin,

rightin,bottomin,leftout,topout,rightout,
                      bottomout)
-----------------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    BOOL active :                --Variable and array
                                     declaration
    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 8:
    INT tag,w,tp,n:
    [size] [size] INT square:
    [size] [size] INT calcul:
    [size] [size] INT temporal:
    [size] INT dummy0:
    [size] INT dummy1:
    [size] INT dummy2:
    [size] INT dummy3:
    [size] INT dummy4:
    [size] INT sender0:
    [size] INT sender1:
    [size] INT sender2:
    [size] INT sender3:
    WHILE TRUE
      SEQ
        SEQ r= 0 FOR size
          SEQ c= 0 FOR size
            SEQ
              square [r] [c] := 0
              calcul [r] [c] := 0
              temporal [r] [c] := 0
        SEQ r= 0 FOR size
          SEQ
            dummy0 [r] := 0
            dummy1 [r] := 0
            dummy2 [r] := 0
            dummy3 [r] := 0
            dummy4 [r] := 0
    sender0 [r] := 0
    sender1 [r] := 0
    sender2 [r] := 0
    sender3 [r] := 0
active:= TRUE
n:= engine
WHILE active
  SEQ
    IF
      n= 5
        SEQ
          leftin ? tag;w
          rightout ! tag;w
          IF
            tag= s
              SEQ
                active:= FALSE
                topout ! square
                bottomin ? temporal
                topout ! temporal
            TRUE
              SEQ
                PAR
                  leftin ?    dummy0
                  topin ?     dummy1
                  rightin ?   dummy2
                  bottomin ?  dummy3
                  leftout !   sender0
                  topout !    sender1
                  rightout !  sender2
                  bottomout ! sender3
                SEQ r = 0 FOR size
                  SEQ
                    square[0] [r] := dummy1 [r]
                    square[r] [0] := dummy0 [r]
                    square[r][size -1] := dummy2 [r]
                    square[size -1][r] := dummy3 [r]
                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
                    SEQ
                      tp:= ((w * square [r] [c] ) +
                           ( square [r] [c-1] +
                           (square [r][c + 1] +
                           ( square [r-1] [c] +
                            square [r + 1] [c] ))))/
                           (4 + w)
                      calcul [r] [c] := tp
                square := calcul
                SEQ r = 0 FOR size
                  SEQ
                    sender0 [r]:= square[r] [1]
                    sender1 [r]:= square[1] [r]
```

```
                                   sender2 [r]:= square[r][size- 2]
                                   sender3 [r]:= square[size-2][r]
  :

----------------------------------------------------------------
 PROC corner.node(VAL INT engine, CHAN OF ANY leftin,topin,
                  rightin,bottomin,leftout,topout,rightout,
                  bottomout)
----------------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    BOOL active :
    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 8:
    INT tag,w,tp,n,counter0:
    [size] [size] INT square:
    [size] [size] INT calcul:
    [size] [size] INT temporal:
    [size] INT dummy0:
    [size] INT dummy1:
    [size] INT dummy2:
    [size] INT dummy3:
    [size] INT dummy4:
    [size] INT sender0:
    [size] INT sender1:
    [size] INT sender2:
    [size] INT sender3:
    WHILE TRUE
      SEQ
        SEQ r= 0 FOR size
          SEQ c= 0 FOR size
            SEQ
              square [r] [c] := 0
              calcul [r] [c] := 0
              temporal [r] [c] := 0
        SEQ r= 0 FOR size
          SEQ
            dummy0 [r] := 0
            dummy1 [r] := 0
            dummy2 [r] := 0
            dummy3 [r] := 0
            dummy4 [r] := 0
            sender0 [r] := 0
            sender1 [r] := 0
            sender2 [r] := 0
            sender3 [r] := 0
        active:= TRUE
        n:= engine
```

```
WHILE active
  SEQ
    IF
      n= 0
        SEQ
          topin ? tag;w;dummy1
          rightout ! tag;w
          bottomout ! tag;w;dummy1
          IF
            tag= s
              SEQ
                counter0:= 0
                active:= FALSE
                topout ! square
                WHILE counter0 < 2
                  SEQ
                    bottomin ? temporal
                    topout ! temporal
                    counter0 := counter0 + 1
                WHILE counter0 < 8
                  SEQ
                    rightin ? temporal
                    topout ! temporal
                    counter0 := counter0 + 1
            TRUE
              SEQ
                PAR
                  leftin ?    dummy0
                  topin ?     dummy4
                  rightin ?   dummy2
                  bottomin ?  dummy3
                  leftout !   sender0
                  topout !    sender1
                  rightout !  sender2
                  bottomout ! sender3
                SEQ r = 0 FOR size
                  SEQ
                    square[r] [0] := dummy1 [r]
                    square[r][size- 1] := dummy2 [r]
                    square[size - 1][r]:= dummy3 [r]
                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
                    SEQ
                      tp:= ((w * square  [r][c]) + (
                                  square [r] [c-1] +
                               ( square [r] [c + 1] + (
                                    square [r-1] [c]  +
                                square [r + 1][c])))) /
                               (4 + w)
                      calcul [r] [c] := tp
                SEQ r = 0 FOR size
                  calcul [r] [0]:= square[r] [0]
                square := calcul
                SEQ r = 0 FOR size
                  SEQ
                    sender0 [r]:= square[r] [1]
                    sender1 [r]:= square[1] [r]
                    sender2 [r]:= square[r][size- 2]
                    sender3 [r]:= square[size-2] [r]

     n= 2
       SEQ
         bottomin ? tag;w;dummy3
         rightout ! tag;w
         IF
           tag= s
             SEQ
               active:= FALSE
               topout ! square
           TRUE
             SEQ
               PAR
                 leftin ?     dummy0
                 topin ?      dummy1
                 rightin ?    dummy2
                 bottomin ?   dummy4
                 leftout !    sender2
                 topout !     sender1
                 rightout !   sender2
                 bottomout !  sender1
               SEQ r = 0 FOR size
                 SEQ
                   square[r] [0] := dummy3 [r]
                   square[0] [r] := dummy1 [r]
                   square[r] [size -1]:= dummy2 [r]
               SEQ r = 1 FOR size - 2
                 SEQ c = 1 FOR size - 2
                   SEQ
                     tp:= ((w * square [r][c] ) + (
                         square [r] [c-1] +
                          ( square  [r] [c + 1] + (
                               square [r-1] [c]  +
                          square [r + 1] [c] )))) /
                          (4 + w)
                     calcul [r] [c] := tp
               SEQ r = 0 FOR size
                 calcul [r] [0] := square [r] [0]
               square := calcul
               SEQ r = 0 FOR size
                 SEQ
                   sender1 [r] := square [1] [r]
                   sender2 [r]:= square[r][size-2]
```

```
n= 8
  SEQ
    leftin ? tag;w
    rightout ! tag
    IF
      tag= s
        SEQ
          active:= FALSE
          leftout ! square
          bottomin ? temporal
          leftout ! temporal
          bottomin ? temporal
          leftout ! temporal
      TRUE
        SEQ
          PAR
            leftin ?     dummy0
            topin ?      dummy1
            rightin ?    dummy2
            bottomin ?   dummy3
            leftout !    sender0
            topout !     sender3
            rightout !   sender0
            bottomout !  sender3
          SEQ r = 0 FOR size
            SEQ
              square[r] [0] := dummy0 [r]
              square[size - 1][r]:= dummy3 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square[r][c]  ) + (
                    square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
                calcul [r] [c] := tp
          square := calcul
          SEQ  r = 0 FOR size
            SEQ
              sender0 [r]:= square [r] [1]
              sender3 [r]:= square[size-2] [r]
```

```
n= 10
  SEQ
    leftin ? tag;w
    IF
      tag= s
        SEQ
          active:= FALSE
          topout ! square
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender1
            rightout !  sender0
            bottomout ! sender1
          SEQ r = 0 FOR size
            SEQ
              square[0] [r] := dummy1 [r]
              square[r] [0] := dummy0 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square [r][c] ) + (
                    square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
                calcul [r] [c] := tp
          square := calcul
          SEQ  r = 0 FOR size
            SEQ
              sender0 [r] := square [r] [1]
              sender1 [r] := square [1] [r]
:
```

```
------------------------------------------------------------------
  PROC cross.node(VAL INT engine, CHAN OF ANY leftin,topin,
                  rightin,bottomin,leftout,topout,
                  rightout,bottomout)
------------------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    BOOL active :
    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 8:
    INT tag,w,tp,n,counter1:
    [size] [size] INT square:
    [size] [size] INT calcul:
    [size] [size] INT temporal:
    [size] INT dummy0:
    [size] INT dummy1:
    [size] INT dummy2:
    [size] INT dummy3:
    [size] INT dummy4:
    [size] INT sender0:
    [size] INT sender1:
    [size] INT sender2:
    [size] INT sender3:
    WHILE TRUE
      SEQ
        SEQ r= 0 FOR size
          SEQ c= 0 FOR size
            SEQ
              square [r] [c] := 0
              calcul [r] [c] := 0
              temporal [r] [c] := 0
        SEQ r= 0 FOR size
          SEQ
            dummy0 [r] := 0
            dummy1 [r] := 0
            dummy2 [r] := 0
            dummy3 [r] := 0
            dummy4 [r] := 0
            sender0 [r] := 0
            sender1 [r] := 0
            sender2 [r] := 0
            sender3 [r] := 0
        active:= TRUE
        n:= engine
```

```
WHILE active
  SEQ
    IF
      n= 1
        SEQ
          topin ? tag;w;dummy1
          rightout ! tag;w
          IF
            tag= s
              SEQ
                active:= FALSE
                topout ! square
                bottomin ? temporal
                topout ! temporal
            TRUE
              SEQ
                PAR
                  leftin ?     dummy0
                  topin ?      dummy4
                  rightin ?    dummy2
                  bottomin ?   dummy3
                  leftout !    sender2
                  topout !     sender1
                  rightout !   sender2
                  bottomout !  sender3
                SEQ r = 0 FOR size
                  SEQ
                    square[r] [0] := dummy1 [r]
                    square[r] [size -1]:= dummy2 [r]
                    square[size - 1][r]:= dummy3 [r]
                    square[0] [r] := dummy4 [r]
                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
                    SEQ
                      tp:= ((w * square[r] [c] ) + (
                          square [r] [c-1] +
                           ( square [r] [c + 1] + (
                            square [r-1] [c]  +
                            square [r + 1][c] )))) /
                           (4 + w)
                      calcul [r] [c] := tp
                SEQ r = 0 FOR size
                  calcul [r] [0]:= square[r] [0]
                square := calcul
                SEQ r = 0 FOR size
                  SEQ
                    sender1 [r]:= square[1] [r]
                    sender2 [r]:= square[r][size -2]
                    sender3 [r]:= square[size-2] [r]
```

```
n= 4
  SEQ
    leftin ? tag;w
    rightout ! tag;w
    IF
      tag= s
        SEQ
          counter1 := 0
          active:= FALSE
          leftout ? square
          WHILE  counter1 < 2
            SEQ
              bottomin ? temporal
              leftout ! temporal
              counter1 := counter1 + 1
          WHILE  counter1 < 5
            SEQ
              rightin ? temporal
              leftout ! temporal
              counter1 := counter1 + 1
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender0
            rightout !  sender2
            bottomout ! sender3
          SEQ r = 0 FOR size
            SEQ
              square[r] [0] := dummy0 [r]
              square[size-1] [r] := dummy3 [r]
              square[r] [size-1] := dummy2 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square[r] [c] ) + (
                     square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
                calcul [r] [c] := tp
          square := calcul
          SEQ r = 0 FOR size
            SEQ
              sender0 [r] := square [r] [1]
              sender2 [r] := square[r][size-2]
              sender3 [r]:= square[size-2] [r]
```

```
n= 6
  SEQ
    leftin ? tag;w
    rightout ! tag;w
    IF
      tag= s
        SEQ
          active:= FALSE
          topout ! square
      TRUE
        SEQ
          PAR
            leftin ?      dummy0
            topin ?       dummy1
            rightin ?     dummy2
            bottomin ?    dummy3
            leftout !     sender0.
            topout !      sender1
            rightout !    sender2
            bottomout !   sender2
          SEQ r = 0 FOR size
            SEQ
              square[r] [0] := dummy0 [r]
              square[0] [r] := dummy1 [r]
              square[r] [size-1] := dummy2 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square[r][c] ) + (
                     square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
                calcul [r] [c] := tp
          square := calcul
          SEQ  r = 0 FOR size
            SEQ
              sender0 [r]:= square [r] [1]
              sender1 [r]:= square [1] [r]
              sender2 [r]:= square [r][size-2]
```

```
n= 9
  SEQ
    leftin ? tag;w
    IF
      tag= s
        SEQ
          active:= FALSE
          topout ! square
          bottomin ? temporal
          topout ! temporal
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender1
            rightout !  sender0
            bottomout ! sender3
          SEQ r = 0 FOR size
            SEQ
              square[size-1][r]:= dummy3 [r]
              square[0] [r] := dummy1 [r]
              square[r] [0] := dummy0 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w  * square[r][c]) + (
                        square [r] [c-1] +
                      ( square [r] [c + 1] + (
                          square [r-1] [c]  +
                      square [r + 1] [c])))) /
                      (4 + w)
                calcul [r] [c] := tp
          square := calcul
          SEQ  r = 0 FOR size
            SEQ
              sender0 [r]:= square [r] [1]
              sender1 [r]:= square [1] [r]
              sender3 [r]:= square [size-2][r]
:
```

```
------------------------------------------------------------
  PROC transp.horizontal (VAL INT engine, CHAN OF ANY
                           leftin,topin,rightin,bottomin,
                           leftout,topout,rightout,
                           bottomout)
------------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    BOOL active:
    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 8:
    INT tag,w,n:
    [size] INT spec1:
    [size] INT spec2:
    WHILE TRUE
      SEQ
        SEQ r = 0 FOR size
          SEQ
            spec1 [r] := 0
            spec2 [r] := 0
        n:= engine
        active:= TRUE
        tag:= g
        WHILE active
          SEQ
            IF
              n= 12
                SEQ
                  leftin ? tag
                  bottomout ! tag
                  IF
                    tag= s
                      active:= FALSE
                    TRUE
                      SEQ
                        leftin ? spec1
                        rightin? spec2
                        leftout ! spec2
                        rightout ! spec1
              n= 13
                SEQ
                  topin ? tag
                  bottomout ! tag
                  IF
                    tag = s
                      active:= FALSE
                    TRUE
                      SEQ
                        leftin ? spec1
                        rightin? spec2
                        leftout ! spec2
                        rightout ! spec1
              n= 14
                SEQ
                  topin ? tag
                  IF
                    tag = s
                      active:= FALSE
                    TRUE
                      SEQ
                        leftin ? spec1
                        rightin? spec2
                        leftout ! spec2
                        rightout ! spec1
 :
-----------------------------------------------------------
 PROC transp.vertical (VAL INT engine, CHAN OF ANY
                       leftin,topin,rightin,bottomin,
                       leftout,topout,rightout,bottomout)
-----------------------------------------------------------
   #USE  "c:\tdsiolib\userio.tsr":
   BOOL active:
   VAL s IS 11:
   VAL g IS 333:
   VAL size IS 8:
   INT tag,w,n:
   [size] INT spec1:
   [size] INT spec2:
   WHILE TRUE
     SEQ
       SEQ r = 0 FOR size
         SEQ
           spec1 [r] := 0
           spec2 [r] := 0
       n:= engine
       active:= TRUE
       tag:= g
       WHILE active
         SEQ
           IF
             n= 3
               SEQ
                 bottomin ? tag;w;spec1
                 topout ! tag;w;spec1
                 rightout ! tag
                 IF
                   tag= s
                     active:= FALSE
                   TRUE
                     SEQ
                       bottomin ? spec1
                       topout ! spec1
                       topin ? spec2
                       bottomout ! spec2
                 n= 7
                   SEQ
                     leftin ? tag
                     rightout ! tag
                     IF
                       tag = s
                         active:= FALSE
                       TRUE
                         SEQ
                           topin ? spec1
                           bottomin? spec2
                           topout ! spec2
                           bottomout ! spec1
                 n= 11
                   SEQ
                     leftin ? tag
                     rightout ! tag
                     IF
                       tag = s
                         active:= FALSE
                       TRUE
                         SEQ
                           topin ? spec1
                           bottomin? spec2
                           topout ! spec2
                           bottomout ! spec1
  :
------------------------------------------------------------
  PROC        neutral.node        (       CHAN       OF       ANY
leftin,topin,rightin,bottomin,
                      leftout,topout,rightout,bottomout)
------------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    BOOL active:
    VAL s IS 11:
    VAL g IS 333:
    INT tag:
    WHILE TRUE
      SEQ
        active:= TRUE
        tag:= g
        WHILE active
          SEQ
            leftin ? tag
            IF
              tag= s
                active:= FALSE
              TRUE
                SEQ
                  SKIP
  :
```

```
------------------------------------------------------------
          -- Processor Placement
------------------------------------------------------------

  PLACED PAR
    PROCESSOR 0 T4
      PLACE channel[0] AT link0in:
      PLACE channel[1] AT link1in:
      PLACE channel[2] AT link2in:
      PLACE channel[3] AT link3in:
      PLACE antichannel[0] AT link0out:
      PLACE antichannel[1] AT link1out:
      PLACE antichannel[2] AT link2out:
      PLACE antichannel[3] AT link3out:
      corner.node(0,channel[0],channel[1],channel[2],
                  channel[3],antichannel[0],antichannel[1],
                  antichannel[2],antichannel[3])

    PROCESSOR 8 T4
      PLACE channel[5] AT link0in:
      PLACE channel[7] AT link1in:
      PLACE channel[8] AT link2in:
      PLACE channel[9] AT link3in:
      PLACE antichannel[5] AT link0out:
      PLACE antichannel[7] AT link1out:
      PLACE antichannel[8] AT link2out:
      PLACE antichannel[9] AT link3out:
      corner.node(8,channel[5],channel[7],channel[8],
                  channel[9],antichannel[5],
                  antichannel[7],antichannel[8],
                  antichannel[9])

    PROCESSOR 2 T4
      PLACE channel[17] AT link0in:
      PLACE channel[12] AT link1in:
      PLACE channel[18] AT link2in:
      PLACE channel[19] AT link3in:
      PLACE antichannel[17] AT link0out:
      PLACE antichannel[12] AT link1out:
      PLACE antichannel[18] AT link2out:
      PLACE antichannel[19] AT link3out:
      corner.node(2,channel[17],channel[12],channel[18],
                  channel[19],antichannel[17],
                  antichannel[12],antichannel[18],
                  antichannel[19])
```

```
PROCESSOR 10 T4
  PLACE channel[20] AT link0in:
  PLACE channel[16] AT link1in:
  PLACE channel[22] AT link2in:
  PLACE channel[23] AT link3in:
  PLACE antichannel[20] AT link0out:
  PLACE antichannel[16] AT link1out:
  PLACE antichannel[22] AT link2out:
  PLACE antichannel[23] AT link3out:
  corner.node(10,channel[20],channel[16],channel[22],
           channel[23],antichannel[20],antichannel[16],
              antichannel[22],antichannel[23])


PROCESSOR 1 T4
  PLACE channel[10] AT link1out:
  PLACE channel[3] AT link2out:
  PLACE channel[11] AT link3out:
  PLACE channel[12] AT link0out:
  PLACE antichannel[10] AT link1in:
  PLACE antichannel[3] AT link2in:
  PLACE antichannel[11] AT link3in:
  PLACE antichannel[12] AT link0in:
  cross.node(1,antichannel[10],antichannel[3],
             antichannel[11],antichannel[12],
             channel[10],channel[3],
             channel[11],channel[12])

PROCESSOR 9 T4
  PLACE channel[13] AT link1out:
  PLACE channel[9] AT link2out:
  PLACE channel[15] AT link3out:
  PLACE channel[16] AT link0out:
  PLACE antichannel[13] AT link1in:
  PLACE antichannel[9] AT link2in:
  PLACE antichannel[15] AT link3in:
  PLACE antichannel[16] AT link0in:
  cross.node(9,antichannel[13],antichannel[9],
             antichannel[15],antichannel[16],
             channel[13],channel[9],
             channel[15],channel[16])
```

```
PROCESSOR 3 T4
  PLACE channel[24] AT link0out:
  PLACE antichannel[30] AT link1out:
  PLACE channel[19] AT link2out:
  PLACE channel[25] AT link3out:
  PLACE antichannel[24] AT link0in:
  PLACE channel[30] AT link1in:
  PLACE antichannel[19] AT link2in:
  PLACE antichannel[25] AT link3in:
  transp.vertical(3,channel[30],antichannel[19],
                  antichannel[25],antichannel[24],
                  antichannel[30],channel[19],
                  channel[25],channel[24])
PROCESSOR 11 T4
  PLACE antichannel[7] AT link0in:
  PLACE channel[26] AT link1in:
  PLACE antichannel[23] AT link2in:
  PLACE antichannel[29] AT link3in:
  PLACE channel[7] AT link0out:
  PLACE antichannel[26] AT link1out:
  PLACE channel[23] AT link2out:
  PLACE channel[29] AT link3out:
  transp.vertical(11,channel[26],antichannel[23],
                  antichannel[29],antichannel[7],
                  antichannel[26],channel[23],
                  channel[29],channel[7])


PROCESSOR 5 T4
  PLACE channel[11] AT link2in:
  PLACE channel[6] AT link3in:
  PLACE channel[13] AT link0in:
  PLACE channel[14] AT link1in:
  PLACE antichannel[11] AT link2out:
  PLACE antichannel[6] AT link3out:
  PLACE antichannel[13] AT link0out:
  PLACE antichannel[14] AT link1out:
  central.node(5,channel[11],channel[6],channel[13],
         channel[14],antichannel[11],antichannel[6],
                    antichannel[13],antichannel[14])
```

```
PROCESSOR 13 T4
  PLACE channel[10] AT link0in:
  PLACE antichannel[28] AT link1in:
  PLACE channel[15] AT link2in:
  PLACE channel[27] AT link3in:
  PLACE antichannel[10] AT link0out:
  PLACE channel[28] AT link1out:
  PLACE antichannel[15] AT link2out:
  PLACE antichannel[27] AT link3out:
  transp.horizontal(13,channel[15],channel[27],
         channel[10],antichannel[28],antichannel[15],
                     antichannel[27],antichannel[10],
                     channel[28])


PROCESSOR 7 T4
  PLACE antichannel[26] AT link0in:
  PLACE channel[4] AT link1in:
  PLACE channel[25] AT link2in:
  PLACE channel[21] AT link3in:
  PLACE channel[26] AT link0out:
  PLACE antichannel[4] AT link1out:
  PLACE antichannel[25] AT link2out:
  PLACE antichannel[21] AT link3out:
  transp.vertical(7,channel[25],channel[21],
                antichannel[26],
                channel[4],antichannel[25],
                antichannel[21],channel[26],
                antichannel[4])

PROCESSOR 15 T4
  PLACE channel[30] AT link0out:
  PLACE channel[32] AT link1out:
  PLACE channel[29] AT link2in:
  PLACE channel[31] AT link3in:
  PLACE antichannel[30] AT link0in:
  PLACE antichannel[32] AT link1in:
  PLACE antichannel[29] AT link2out:
  PLACE antichannel[31] AT link3out:
  neutral.node(channel[29],channel[31],channel[30],
         channel[32],antichannel[29],antichannel[31],
         antichannel[30],antichannel[32])
```

```
PROCESSOR 4 T4
  PLACE channel[2] AT link3out:
  PLACE channel[4] AT link0out:
  PLACE channel[5] AT link1out:
  PLACE channel[6] AT link2out:
  PLACE antichannel[2] AT link3in:
  PLACE antichannel[4] AT link0in:
  PLACE antichannel[5] AT link1in:
  PLACE antichannel[6] AT link2in:
  cross.node(4,antichannel[2],antichannel[4],
        antichannel[5],antichannel[6],channel[2],
        channel[4],channel[5],channel[6])


PROCESSOR 6 T4
  PLACE channel[18] AT link3out:
  PLACE channel[14] AT link0out:
  PLACE channel[20] AT link1out:
  PLACE channel[21] AT link2out:
  PLACE antichannel[18] AT link3in:
  PLACE antichannel[14] AT link0in:
  PLACE antichannel[20] AT link1in:
  PLACE antichannel[21] AT link2in:
  cross.node(6,antichannel[18],antichannel[14],
     antichannel[20],antichannel[21],channel[18],
             channel[14],channel[20],channel[21])


PROCESSOR 12 T4
  PLACE channel[32] AT link0in:
  PLACE antichannel[0] AT link1in:
  PLACE antichannel[27] AT link2in:
  PLACE antichannel[8] AT link3in:
  PLACE antichannel[32] AT link0out:
  PLACE channel[0] AT link1out:
  PLACE channel[27] AT link2out:
  PLACE channel[8] AT link3out:
  transp.horizontal(12,antichannel[8],channel[32],
              antichannel[0],antichannel[27],channel[8],
              antichannel[32],channel[0],channel[27])
```

```
PROCESSOR 14 T4
  PLACE channel[28] AT link0in:
  PLACE antichannel[17] AT link1in:
  PLACE antichannel[31] AT link2in:
  PLACE antichannel[22] AT link3in:
  PLACE antichannel[28] AT link0out:
  PLACE channel[17] AT link1out:
  PLACE channel[31] AT link2out:
  PLACE channel[22] AT link3out:
  transp.horizontal(14,antichannel[22],channel[28],
                    antichannel[17],antichannel[31],
                channel[22],antichannel[28],channel[17],
                    channel[31])
```

APPENDIX D

16 TRANSPUTER NETWORK SOURCE CODE

```
------------------------------------------------------------
-----
  PROC input.handler (CHAN OF ANY keyboard,screen)

  -- This   procedure  send  the  boundary  conditions  to
processors  0   and 3
  -- on the network and display the information  coming from
the
  network
  -- when it stops the network.
-------------------------------------------------------------
                             -- Channel and link decla.
    #USE  "c:\tdsiolib\userio.tsr":
    VAL link0out  IS 0:
    VAL link1out  IS 1:
    VAL link2out  IS 2:
    VAL link3out  IS 3:
    VAL link0in   IS 4:
    VAL link1in   IS 5:
    VAL link2in   IS 6:
    VAL link3in   IS 7:
    CHAN OF ANY leftin,rightout,antirightout,antileftin:
    PLACE leftin AT link3in:
    PLACE rightout  AT link3out:              -- placement of
    PLACE antirightout AT link2out:    -- external channels
    PLACE antileftin AT link2in:
    VAL s IS 11:
    VAL esc IS 223:
    VAL g IS 333:
    VAL size IS 6:
    [size] INT temp:              -- Array declarations
    [size] INT recp:
    [size] INT recp1:
    [size] INT recp2:
    [size] [size] INT truly:
    [16][size] [size] INT true:
    BOOL turning:
    INT w,tag,he,no,z,counter,counter1,txt:
    SEQ
      no:=0
      write.full.string (screen, " Enter the hot end
                   temperature")
      read.echo.int (keyboard,screen,he,no)
      newline(screen)
      no:=0
      write.full.string (screen, " Enter the propagation
                              rate ")
      read.echo.int (keyboard,screen,w,no)
      newline(screen)
```

```
SEQ
  SEQ r = 0 FOR size         -- Initialization of
    SEQ                      -- arrays
      temp [r] := 0
      recp [r] := 0
      recp1 [r] := 0
      recp2 [r] := 0
  SEQ r = 0 FOR size
    temp [r] := he
  tag:= g
  antirightout ! tag;w;temp  -- sending hot end and W
  rightout ! tag;w;temp        -- and start signal
  antirightout ! recp2
  rightout ! recp1
  turning := TRUE
  SEQ
    WHILE turning
      PRI ALT
        keyboard ?  z          -- receive stop signal
               SEQ
            IF
              z = esc
                SEQ
                  SEQ
                    tag:= s
                    antileftin ? recp
                    leftin ? recp
                    antirightout ! tag;w;temp
                    rightout ! tag;w;temp
                    counter := 0
                    counter1 := 0
                    WHILE counter < 16 -- receiving
                      SEQ                     -- arrays
                        antileftin ? truly
                        SEQ h = 0 FOR size
                          SEQ p = 0 FOR size
                            true [counter]  [h] [p] :=
                                    truly [h] [p]
                        counter := counter + 1
                    SEQ
                      SEQ r = 0 FOR size - 1
                        SEQ
                          SEQ c = 0 FOR size - 1
                            SEQ
                              txt:= true  [counter1]
                                         [r] [c]
                              write.int (screen,txt,3)
                          SEQ l = 1 FOR size - 2
                            SEQ
                              txt:= true  [counter1 +
                                        4] [r]  [l]
                              write.int (screen,txt,3)
        SEQ f = 1 FOR size - 2
          SEQ
            txt:= true  [counter1 +
                     8] [r][f]
            write.int (screen,txt,3)
        SEQ d = 1 FOR size - 1
          SEQ
            txt:= true [counter1+12]
                              [r] [d]
            write.int (screen,txt,3)
        newline(screen)
    counter1:= counter1 + 1
    SEQ r = 1 FOR size - 2
      SEQ
        SEQ c = 0 FOR size - 1
          SEQ
            txt:= true  [counter1]
                           [r] [c]
            write.int (screen,txt,3)
        SEQ l = 1 FOR size - 2
          SEQ
            txt:= true[counter1+4]
                              [r][l]
            write.int (screen,txt,3)
        SEQ f = 1 FOR size - 2
          SEQ
            txt:= true  [counter1 +
                           8] [r][f]
            write.int (screen,txt,3)
        SEQ d = 1 FOR size - 1
          SEQ
            txt:= true [counter1+12]
                                 [r][d]
            write.int (screen,txt,3)
        newline(screen)
    counter1:= counter1 + 1
    SEQ r = 1 FOR size - 2
      SEQ
        SEQ c = 0 FOR size - 1
          SEQ
            txt:= true  [counter1]
                           [r] [c]
            write.int (screen,txt,3)
        SEQ l = 1 FOR size - 2
          SEQ
            txt:= true  [counter1 +
                         4] [r][l]
            write.int (screen,txt,3)
        SEQ f = 1 FOR size - 2
          SEQ
            txt:= true  [counter1 +
                        8] [r] [f]
```

```
                              write.int (screen,txt,3)
                          SEQ d = 1 FOR size - 1
                            SEQ
                              txt:= true  [counter1 +
                                       12][r][d]
                              write.int (screen,txt,3)
                          newline(screen)
                      counter1:= counter1 + 1
                      SEQ r = 1 FOR size - 1
                        SEQ
                          SEQ c = 0 FOR size - 1
                            SEQ
                              txt:= true  [counter1]
                                          [r] [c]
                              write.int (screen,txt,3)
                          SEQ l = 1 FOR size - 2
                            SEQ
                              txt:= true  [counter1 +
                                          4] [r] [l]
                              write.int (screen,txt,3)
                          SEQ f = 1 FOR size - 2
                            SEQ
                              txt:= true[counter1+8]
                                             [r] [f]
                              write.int (screen,txt,3)
                          SEQ d = 1 FOR size - 1
                            SEQ
                              txt:= true [counter1+12]
                                                [r][d]
                              write.int (screen,txt,3)
                          newline(screen)
                    turning := FALSE
                  newline(screen)
        antileftin ? recp1
          SEQ
            leftin ? recp2
            antirightout ! tag;w;temp
            rightout ! tag;w;temp
            antirightout ! recp2
            rightout ! recp1
newline(screen)
write.full.string(screen, "Type ANY to return to TDS")
INT any:
read.char(keyboard, any)
```

```
------------------------------------------------------------
  -- variables and channel declarations
  --------------------------------------------------------
-----  VAL link0out  IS 0:
  VAL link1out  IS 1:
  VAL link2out  IS 2:
  VAL link3out  IS 3:
  VAL link0in   IS 4:
  VAL link1in   IS 5:
  VAL link2in   IS 6:
  VAL link3in   IS 7:
  [33] CHAN OF ANY channel,antichannel:
-------------------------------------------------------------

  PROC central.node(VAL INT engine, CHAN OF ANY
                    leftin,topin,rightin,bottomin,
                    leftout,topout,rightout,bottomout)

  -- This  procedure does  the calculations for nodes at the
center    -- of the network
 -------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
    -----------------------------------------------------
-----
    -- Declarations of arrays and variables
    -----------------------------------------------------
-----    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 6:
    [size] [size] INT square:
    [size] [size] INT calcul:
    [size] INT dummy0:
    [size] INT dummy1:
    [size] INT dummy2:
    [size] INT dummy3:
    [size] INT dummy4:
    [size] INT sender0:
    [size] INT sender1:
    [size] INT sender2:
    [size] INT sender3:
    [size] [size] INT temporal:
    BOOL active :
    INT tag,w,tp,n:
```

```
WHILE TRUE
  SEQ
    SEQ r= 0 FOR size      -- Initialization of arrays
      SEQ c= 0 FOR size
        SEQ
          square [r] [c] := 0
          calcul [r] [c] := 0
          temporal [r] [c] := 0
    SEQ r= 0 FOR size
      SEQ
        dummy0 [r] := 0
        dummy1 [r] := 0
        dummy2 [r] := 0
        dummy3 [r] := 0
        dummy4 [r] := 0
        sender0 [r] := 0
        sender1 [r] := 0
        sender2 [r] := 0
        sender3 [r] := 0
    active:= TRUE
    n:= engine
    WHILE active
      SEQ
        IF
          (n= 5) OR (n= 9)-- code for processors 5 and 9
           SEQ
              leftin ? tag;w --receiving start/stop
              rightout ! tag;w  -- sending start/stop
              IF
                tag= s
                  SEQ
                    active:= FALSE -- checking for stop
                    topout ! square  -- routing code to
                    bottomin ? temporal
                    topout ! temporal
                    bottomin ? temporal
                    topout ! temporal
                TRUE
                  SEQ   --  Communications receive
                    PAR     --  send boundaries
                 --    conditions
                      leftin ?    dummy0
                      topin ?     dummy1
                      rightin ?   dummy2
                      bottomin ?  dummy3
                      leftout !   sender0
                      topout !    sender1
                      rightout !  sender2
                      bottomout ! sender3
        SEQ r = 0 FOR size
               SEQ
            square[0] [r] := dummy1 [r]
            square[r] [0] := dummy0 [r]
            square[r] [size - 1] := dummy2 [r]
            square[size - 1] [r] := dummy3 [r]
        SEQ r = 1 FOR size - 2
              SEQ c = 1 FOR size - 2
                 SEQ
                tp:= ((w * square [r][c] ) + (
                    square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
                calcul [r] [c] := tp
          square := calcul
          SEQ r = 0 FOR size
            SEQ
              sender0 [r]:= square[r] [1]
              sender1 [r]:= square[1] [r]
              sender2 [r]:= square[r][size- 2]
              sender3 [r]:= square[size-2] [r]
(n= 6) OR (n= 10) -- code processors 6 and 10
  SEQ             -- in the network
    leftin ? tag;w
    rightout ! tag;w
    IF
      tag= s
        SEQ                    -- checking stop
          active:= FALSE
          topout ! square  -- routing code
          bottomin ? temporal
          topout ! temporal
      TRUE
        SEQ
          PAR     -- COMMUNICATIONS BLOCK
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender1
            rightout !  sender2
            bottomout ! sender3
          SEQ r = 0 FOR size
            SEQ
              square[0] [r] := dummy1 [r]
              square[r] [0] := dummy0 [r]
              square[r] [size-1] := dummy2 [r]
              square[size -1][r] := dummy3 [r]
          SEQ r = 1 FOR size - 2
```

```
                                     SEQ c = 1 FOR size - 2
                                    SEQ
                                      tp:= ((w * square [r][c] ) + (
                                          square [r] [c-1] +
                                           ( square [r] [c + 1] + (
                                           square [r-1] [c]  +
                                           square [r + 1] [c] )))) /
                                           (4 + w)
                                      calcul [r] [c] := tp
                                square := calcul
                                SEQ r = 0 FOR size
                                  SEQ
                                    sender0 [r]:= square[r] [1]
                                    sender1 [r]:= square[1] [r]
                                    sender2 [r]:= square[r][size- 2]
                                    sender3 [r]:= square[size- 2][r]
  :
----------------------------------------------------------
  PROC corner.node(VAL INT engine, CHAN OF ANY
     leftin,topin,rightin,bottomin,
                   leftout,topout,rightout,bottomout)

    -- This procedure drives the execution of the processors
       at the corners
    -- of the array

----------------------------------------------------------

    #USE  "c:\tdsiolib\userio.tsr":
    ------------------------------------------------------
       -- declarations of arrays and variables
    ------------------------------------------------------
    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 6:
    [size] [size] INT square:
    [size] [size] INT calcul:
    [size] [size] INT temporal:
    [size] INT dummy0:
    [size] INT dummy1:
    [size] INT dummy2:
    [size] INT dummy3:
    [size] INT dummy4:
    [size] INT sender0:
    [size] INT sender1:
    [size] INT sender2:
    [size] INT sender3:
    BOOL active :
    INT tag,w,tp,n,counter0:
    WHILE TRUE
```

```
SEQ
  SEQ r= 0 FOR size     -- Initialization of arrays
    SEQ c= 0 FOR size
      SEQ
        square [r] [c] := 0
        calcul [r] [c] := 0
        temporal [r] [c] := 0
  SEQ r= 0 FOR size
    SEQ
      dummy0 [r] := 0
      dummy1 [r] := 0
      dummy2 [r] := 0
      dummy3 [r] := 0
      dummy4 [r] := 0
      sender0 [r] := 0
      sender1 [r] := 0
      sender2 [r] := 0
      sender3 [r] := 0
  active:= TRUE
  n:= engine
  WHILE active
    SEQ
      IF
        n= 0               -- code for processor 0
          SEQ
            topin ? tag;w;dummy1
            rightout ! tag;w
            bottomout ! tag;w;dummy1
            IF
              tag= s
                SEQ    -- checking for stop
                  counter0:= 0
                  active:= FALSE
                  topout ! square
                  WHILE counter0 < 3
                     SEQ    -- screen array information
                      bottomin ? temporal
                      topout ! temporal
                      counter0 := counter0 + 1
                  WHILE counter0 < 15
                         SEQ
                                rightin ? temporal
                      topout ! temporal
                      counter0 := counter0 + 1
              TRUE
                SEQ
                  PAR
                    leftin ?    dummy0
                    topin ?     dummy4
                    rightin ?   dummy2
                    bottomin ?  dummy3
                    leftout !   sender0
                  topout !    sender1
                  rightout !  sender2
                  bottomout ! sender3
                SEQ r = 0 FOR size
                  SEQ
                   square[r] [0] := dummy1 [r]
                   square[r] [size - 1] := dummy2[r]
                   square[size - 1] [r] := dummy3[r]
                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
        SEQ
                      tp:= ((w * square [r] [c] ) +(
                          square [r] [c-1] +
                           ( square [r] [c + 1] + (
                           square [r-1] [c]  +
                           square [r + 1] [c] )))) /
                           (4 + w)
                      calcul [r] [c] := tp
                SEQ r = 0 FOR size
                  calcul [r] [0]:= square[r] [0]
                square := calcul
                SEQ r = 0 FOR size
                   SEQ
                     sender0 [r]:= square[r] [1]
                     sender1 [r]:= square[1] [r]
                     sender2 [r]:= square[r][size-2]
                     sender3 [r]:= square[size-2][r]

     n= 3              -- code for processor 3
        SEQ
          bottomin ? tag;w;dummy3
          topout ! tag;w;dummy3
          rightout ! tag;w
          IF
            tag= s
              SEQ
                active:= FALSE
                topout ! square
            TRUE
              SEQ
                PAR
                  leftin ?    dummy0
                  topin ?     dummy1
                  rightin ?   dummy2
                  bottomin ?  dummy4
                  leftout !   sender2
                  topout !    sender1
                  rightout !  sender2
                  bottomout ! sender1

                SEQ r = 0 FOR size
              SEQ
                square[r] [0] := dummy3 [r]
                square[0] [r] := dummy1 [r]
                square[r] [size -1]:= dummy2 [r]
            SEQ r = 1 FOR size - 2
              SEQ c = 1 FOR size - 2
                SEQ
                  tp:= ((w * square[r][c])  + (
                     square [r] [c-1] +
                       ( square [r] [c + 1] + (
                       square [r-1] [c]  +
                       square [r + 1] [c] )))) /
                       (4 + w)
                  calcul [r] [c] := tp
            SEQ r = 0 FOR size
              calcul [r] [0] := square [r] [0]
            square := calcul
            SEQ r = 0 FOR size
              SEQ
                sender1 [r] := square[1] [r]
                sender2 [r] := square[r][size-2]

  n= 12   -- code for processor 12
    SEQ
      leftin ? tag;w
      IF
        tag= s
          SEQ
            counter0 := 0
            active:= FALSE
            leftout ! square
            WHILE counter0 < 3
              SEQ
                bottomin ? temporal
                leftout ! temporal
                counter0:= counter0 + 1
        TRUE
          SEQ
            PAR
              leftin ?    dummy0
              topin ?     dummy1
              rightin ?   dummy2
              bottomin ?  dummy3
              leftout !   sender0
              topout !    sender3
              rightout !  sender0
              bottomout ! sender3
```

```
                  SEQ r = 0 FOR size
                    SEQ
                      square[r] [0] := dummy0 [r]
                      square[size - 1][r]:= dummy3 [r]
                  SEQ r = 1 FOR size - 2
                    SEQ c = 1 FOR size - 2
                      SEQ
                        tp:= ((w * square [r][c] ) + (
                             square [r] [c-1] +
                              ( square [r] [c + 1] + (
                              square [r-1] [c]  +
                              square [r + 1] [c] ))))/
                              (4 + w)
                        calcul [r] [c] := tp
                  square := calcul
                  SEQ  r = 0 FOR size
                    SEQ
                      sender0 [r] :=square [r] [1]
                      sender3 [r] :=square [size-2][r]


n= 15    -- code for processor 15
  SEQ
    leftin ? tag;w
    IF
      tag= s
        SEQ
          active:= FALSE
          topout ! square
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender1
            rightout !  sender0
            bottomout ! sender1
          SEQ r = 0 FOR size
            SEQ
              square[0] [r] := dummy1 [r]
              square[r] [0] := dummy0 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square [r][c] ) + (
                    square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
```

```
                                  (4 + w)
                              calcul [r] [c] := tp
                        square := calcul
                        SEQ  r = 0 FOR size
                          SEQ
                            sender0 [r] := square [r] [1]
                            sender1 [r] := square [1] [r]
  :
---------------------------------------------------------
  PROC cross.node(VAL  INT engine, CHAN OF ANY
                  leftin,topin,rightin,bottcmin,
               leftout,topout,rightout,bottomout)

      -- This procedure drives the processors which are
          situated
      -- forming a
      -- croos at the square network
-----------------------------------------------------------
    #USE  "c:\tdsiolib\userio.tsr":
--------------------------------------------------------
        -- declarations of arrays, variables and constant

-------------------------------------------------------------
    VAL s IS 11:
    VAL g IS 333:
    VAL size IS 6:
    [size] [size] INT square:
    [size] [size] INT calcul:
    [size] [size] INT temporal:
    [size] INT dummy0:
    [size] INT dummy1:
    [size] INT dummy2:
    [size] INT dummy3:
    [size] INT dummy4:
    [size] INT sender0:
    [size] INT sender1:
    [size] INT sender2:
    [size] INT sender3:
    BOOL active :
    INT tag,w,tp,n,counter1:
    WHILE TRUE
      SEQ
        SEQ r= 0 FOR size   -- Initialization of arrays
          SEQ c= 0 FOR size
            SEQ
              square [r] [c] := 0
              calcul [r] [c] := 0
              temporal [r] [c] := 0
        SEQ r= 0 FOR size
          SEQ
            dummy0 [r] := 0
            dummy1 [r] := 0
          dummy2 [r] := 0
          dummy3 [r] := 0
          dummy4 [r] := 0
          sender0 [r] := 0
          sender1 [r] := 0
          sender2 [r] := 0
          sender3 [r] := 0
      active:= TRUE
      n:= engine
      WHILE active
        SEQ
          IF
            n= 1          -- code for processor 01
              SEQ
                topin ? tag;w;dummy1
                rightout ! tag;w
                  IF   -- sending start/stop signal
                    tag= s
                      SEQ    -- checking for stop
                        active:= FALSE
                      topout ! square -- routing code
                 bottomin ? temporal
                      topout ! temporal
                      bottomin ? temporal
                      topout ! temporal
                  TRUE
                    SEQ
                      PAR
                        leftin ?    dummy0
                        topin ?     dummy4
                        rightin ?   dummy2
                        bottomin ?  dummy3
                        leftout !   sender2
                        topout !    sender1
                        rightout !  sender2
                        bottomout ! sender3
                      SEQ r = 0 FOR size
                        SEQ
                          square[r] [0] := dummy1 [r]
                          square[r] [size -1]:= dummy2 [r]
                          square[size - 1][r]:= dummy3 [r]
                          square[0] [r] := dummy4 [r]




                      SEQ r = 1 FOR size - 2
                        SEQ c = 1 FOR size - 2
                           SEQ
                    tp:= ((w * square [r][c] ) + (
                        square [r] [c-1] +
                         ( square [r] [c + 1] + (
                         square [r-1] [c]  +
                         square [r + 1] [c] )))) /
                         (4 + w)
                    calcul [r] [c] := tp
              SEQ r = 0 FOR size
                calcul [r] [0]:= square[r] [0]
              square := calcul
              SEQ r = 0 FOR size
                SEQ
                  sender1 [r]:= square[1] [r]
                  sender2 [r]:= square[r][size-2]
                  sender3 [r]:= square[size-2] [r]

n= 2            -- code for processor 2
  SEQ
    bottomin ? tag;w;dummy1
    rightout ! tag;w
    IF
      tag= s
        SEQ
          active:= FALSE
          topout ! square
          bottomin ? temporal
          topout ! temporal
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy4
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender2
            topout !    sender1
            rightout !  sender2
            bottomout ! sender3
          SEQ r = 0 FOR size
            SEQ
              square[r] [0] := dummy1 [r]
              square[r] [size-1] := dummy2 [r]
              square[size - 1][r] := dummy3[r]
              square[0] [r] := dummy4 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square [r][c]) + (
                     square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
                calcul [r] [c] := tp
          SEQ r = 0 FOR size
            calcul [r] [0]:= square[r] [0]
          square := calcul
          SEQ r = 0 FOR size
            SEQ
              sender1 [r]:= square[1] [r]
              sender2 [r]:= square[r][size- 2]
              sender3 [r]:= square[size-2][r]

n= 4     -- code for processor 4
  SEQ
    leftin ? tag;w
    rightout ! tag;w
    IF
      tag= s
        SEQ
          counter1 := 0
          active:= FALSE
          leftout ? square
          WHILE  counter1 < 3
            SEQ
              bottomin ? temporal
              leftout ! temporal
              counter1 := counter1 + 1
          WHILE  counter1 < 11
            SEQ
              rightin ? temporal
              leftout ! temporal
              counter1 := counter1 + 1
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender0
            rightout !  sender2
            bottomout ! sender3



          SEQ r = 0 FOR size
            SEQ
              square[r] [0] := dummy0 [r]
              square[size -1][r] := dummy3 [r]
              square[r] [size-1] := dummy2 [r]
```

```
                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
                    SEQ
                      tp:= ((w * square[r] [c] ) + (
                          square [r] [c-1] +
                           ( square [r] [c + 1] + (
                           square [r-1] [c]  +
                           square [r + 1] [c] )))) /
                           (4 + w)
                      calcul [r] [c] := tp
                square := calcul
                SEQ r = 0 FOR size
                  SEQ
                    sender0 [r] := square[r][1]
                    sender2 [r] := square[r][size-2]
                    sender3 [r] := square[size-2][r]

      n= 8        -- code for processor 8
        SEQ
          leftin ? tag;w
          rightout ! tag;w
          IF
            tag= s
              SEQ
                counter1 := 0
                active:= FALSE
                leftout ? square
                WHILE  counter1 < 3
                  SEQ
                    bottomin ? temporal
                    leftout ! temporal
                    counter1 := counter1 + 1
                WHILE  counter1 < 7
                  SEQ
                    rightin ? temporal
                    leftout ! temporal
                    counter1 := counter1 + 1
            TRUE
              SEQ
                PAR
                  leftin ?     dummy0
                  topin ?      dummy1
                  rightin ?    dummy2
                  bottomin ?   dummy3
                  leftout !    sender0
                  topout !     sender0
                  rightout !   sender2
                  bottomout !  sender3
                SEQ r = 0 FOR size
                  SEQ
                    square[r] [0] := dummy0 [r]
                    square[size - 1][r]:= dummy3 [r]
                    square[r][size -1] := dummy2 [r]
                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
                    SEQ
                      tp:= ((w * square[r][c] ) + (
                           square [r] [c-1] +
                           ( square [r] [c + 1] + (
                           square [r-1] [c]  +
                           square [r + 1] [c] )))) /
                           (4 + w)
                      calcul [r] [c] := tp
                square := calcul
                SEQ r = 0 FOR size
                  SEQ
                    sender0 [r] := square[r] [1]
                    sender2 [r] := square[r][size-2]
                    sender3 [r] := square[size-2][r]

      (n= 7) OR (n= 11) -- code processor 7 and 11
        SEQ
          leftin ? tag;w
          rightout ! tag;w
          IF
            tag= s
              SEQ
                active:= FALSE
                topout ! square
            TRUE
              SEQ
                PAR
                  leftin ?    dummy0
                  topin ?     dummy1
                  rightin ?   dummy2
                  bottomin ?  dummy3
                  leftout !   sender0
                  topout !    sender1
                  rightout !  sender2
                  bottomout ! sender2
                SEQ r = 0 FOR size
                  SEQ
                    square[r] [0] := dummy0 [r]
                    square[0] [r] := dummy1 [r]
                    square[r] [size- 1]:= dummy2[r]




                SEQ r = 1 FOR size - 2
                  SEQ c = 1 FOR size - 2
                    SEQ
                    tp:= ((w * square [r] [c])+ (
                          square [r] [c-1] +
                         ( square [r] [c + 1] + (
                         square [r-1] [c]  +
                         square [r + 1] [c] )))) /
                         (4 + w)
                    calcul [r] [c] := tp
              square := calcul
              SEQ  r = 0 FOR size
                SEQ
                  sender0 [r] := square [r] [1]
                  sender1 [r] := square [1] [r]
                  sender2 [r] := square[r][size-2]

n= 13      -- code for processor  13
  SEQ
    leftin ? tag;w
    IF
      tag= s
        SEQ
          active:= FALSE
          topout ! square
          bottomin ? temporal
          topout ! temporal
          bottomin ? temporal
          topout ! temporal

      TRUE
        SEQ
          PAR
            leftin ?     dummy0
            topin ?      dummy1
            rightin ?    dummy2
            bottomin ?   dummy3
            leftout !    sender0
            topout !     sender1
            rightout !   sender0
            bottomout !  sender3
          SEQ r = 0 FOR size
            SEQ
              square[0] [r] := dummy1 [r]
              square[r] [0] := dummy0 [r]
              square[size - 1][r]:= dummy3 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square [r][c]) + (
                     square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
```

```
                 calcul [r] [c] := tp
           square := calcul
           SEQ  r = 0 FOR size
             SEQ
               sender0 [r] := square [r] [1]
               sender1 [r] := square [1] [r]
               sender3 [r] := square[size-2][r]

n= 14          -- code for processor 14
  SEQ
    leftin ? tag;w
    IF
      tag= s
        SEQ
          active:= FALSE
          topout ! square
          bottomin ? temporal
          topout ! temporal
      TRUE
        SEQ
          PAR
            leftin ?    dummy0
            topin ?     dummy1
            rightin ?   dummy2
            bottomin ?  dummy3
            leftout !   sender0
            topout !    sender1
            rightout !  sender0
            bottomout ! sender3
          SEQ r = 0 FOR size
            SEQ
              square[0] [r] := dummy1 [r]
              square[r] [0] := dummy0 [r]
              square[size - 1][r]:= dummy3 [r]
          SEQ r = 1 FOR size - 2
            SEQ c = 1 FOR size - 2
              SEQ
                tp:= ((w * square [r][c] ) + (
                     square [r] [c-1] +
                     ( square [r] [c + 1] + (
                     square [r-1] [c]  +
                     square [r + 1] [c] )))) /
                     (4 + w)
                calcul [r] [c] := tp
          square := calcul

          SEQ  r = 0 FOR size
            SEQ
              sender0 [r] := square [r] [1]
              sender1 [r] := square [1] [r]
              sender3 [r] := square[size-2][r]
:
```

```
-----------------------------------------------------------

          Placement of the processors
-----------------------------------------------------------

  PLACED PAR
    PROCESSOR 0 T4
      PLACE channel[0] AT link0in:
      PLACE channel[1] AT link1in:
      PLACE channel[2] AT link2in:
      PLACE channel[3] AT link3in:
      PLACE antichannel[0] AT link0out:
      PLACE antichannel[1] AT link1out:
      PLACE antichannel[2] AT link2out:
      PLACE antichannel[3] AT link3out:
      corner.node(0,channel[0],channel[1],channel[2],
                  channel[3],antichannel[0],antichannel[1],
                  antichannel[2],antichannel[3])

    PROCESSOR 8 T4
      PLACE channel[5] AT link0in:
      PLACE channel[7] AT link1in:
      PLACE channel[8] AT link2in:
      PLACE channel[9] AT link3in:
      PLACE antichannel[5] AT link0out:
      PLACE antichannel[7] AT link1out:
      PLACE antichannel[8] AT link2out:
      PLACE antichannel[9] AT link3out:
      cross.node(8,channel[5],channel[7],channel[8],
           channel[9],antichannel[5],antichannel[7],
                      antichannel[8],antichannel[9])

    PROCESSOR 2 T4
      PLACE channel[17] AT link0in:
      PLACE channel[12] AT link1in:
      PLACE channel[18] AT link2in:
      PLACE channel[19] AT link3in:
      PLACE antichannel[17] AT link0out:
      PLACE antichannel[12] AT link1out:
      PLACE antichannel[18] AT link2out:
      PLACE antichannel[19] AT link3out:
      cross.node(2,channel[17],channel[12],channel[18],
           channel[19],antichannel[17],antichannel[12],
                       antichannel[18],antichannel[19])
```

```
PROCESSOR 10 T4
  PLACE channel[20] AT link0in:
  PLACE channel[16] AT link1in:
  PLACE channel[22] AT link2in:
  PLACE channel[23] AT link3in:
  PLACE antichannel[20] AT link0out:
  PLACE antichannel[16] AT link1out:
  PLACE antichannel[22] AT link2out:
  PLACE antichannel[23] AT link3out:
  central.node(10,channel[20],channel[16],channel[22],
          channel[23],antichannel[20],antichannel[16],
                      antichannel[22],antichannel[23])

PROCESSOR 1 T4
  PLACE channel[10] AT link1out:
  PLACE channel[3] AT link2out:
  PLACE channel[11] AT link3out:
  PLACE channel[12] AT link0out:
  PLACE antichannel[10] AT link1in:
  PLACE antichannel[3] AT link2in:
  PLACE antichannel[11] AT link3in:
  PLACE antichannel[12] AT link0in:
  cross.node(1,antichannel[10],antichannel[3],
  antichannel[11],antichannel[12],channel[10],
           channel[3],channel[11],channel[12])

PROCESSOR 9 T4
  PLACE channel[13] AT link1out:
  PLACE channel[9] AT link2out:
  PLACE channel[15] AT link3out:
  PLACE channel[16] AT link0out:
  PLACE antichannel[13] AT link1in:
  PLACE antichannel[9] AT link2in:
  PLACE antichannel[15] AT link3in:
  PLACE antichannel[16] AT link0in:
  central.node(9,antichannel[13],antichannel[9],
    antichannel[15],antichannel[16],channel[13],
             channel[9],channel[15],channel[16])
```

```
PROCESSOR 3 T4
  PLACE channel[24] AT link0out:
  PLACE antichannel[30] AT link1out:
  PLACE channel[19] AT link2out:
  PLACE channel[25] AT link3out:
  PLACE antichannel[24] AT link0in:
  PLACE channel[30] AT link1in:
  PLACE antichannel[19] AT link2in:
  PLACE antichannel[25] AT link3in:
  corner.node(3,channel[30],antichannel[19],
            antichannel[25],antichannel[24],
            antichannel[30],channel[19],
            channel[25],channel[24])

PROCESSOR 11 T4
  PLACE antichannel[7] AT link0in:
  PLACE channel[26] AT link1in:
  PLACE antichannel[23] AT link2in:
  PLACE antichannel[29] AT link3in:
  PLACE channel[7] AT link0out:
  PLACE antichannel[26] AT link1out:
  PLACE channel[23] AT link2out:
  PLACE channel[29] AT link3out:
  cross.node(11,channel[26],antichannel[23],
             antichannel[29],antichannel[7],
    antichannel[26],channel[23],channel[29],
                                 channel[7])

PROCESSOR 5 T4
  PLACE channel[11] AT link2in:
  PLACE channel[6] AT link3in:
  PLACE channel[13] AT link0in:
  PLACE channel[14] AT link1in:
  PLACE antichannel[11] AT link2out:
  PLACE antichannel[6] AT link3out:
  PLACE antichannel[13] AT link0out:
  PLACE antichannel[14] AT link1out:
  central.node(5,channel[11],channel[6],channel[13],
         channel[14],antichannel[11],antichannel[6],
         antichannel[13],antichannel[14])
```

```
PROCESSOR 13 T4
  PLACE channel[10] AT link0in:
  PLACE antichannel[28] AT link1in:
  PLACE channel[15] AT link2in:
  PLACE channel[27] AT link3in:
  PLACE antichannel[10] AT link0out:
  PLACE channel[28] AT link1out:
  PLACE antichannel[15] AT link2out:
  PLACE antichannel[27] AT link3out:
  cross.node(13,channel[15],channel[27],channel[10],
             antichannel[28],antichannel[15],
             antichannel[27],antichannel[10],
             channel[28])

PROCESSOR 7 T4
  PLACE antichannel[26] AT link0in:
  PLACE channel[4] AT link1in:
  PLACE channel[25] AT link2in:
  PLACE channel[21] AT link3in:
  PLACE channel[26] AT link0out:
  PLACE antichannel[4] AT link1out:
  PLACE antichannel[25] AT link2out:
  PLACE antichannel[21] AT link3out:
  cross.node(7,channel[25],channel[21],antichannel[26],
             channel[4],antichannel[25],antichannel[21],
             channel[26],antichannel[4])

PROCESSOR 15 T4
  PLACE channel[30] AT link0out:
  PLACE channel[32] AT link1out:
  PLACE channel[29] AT link2in:
  PLACE channel[31] AT link3in:
  PLACE antichannel[30] AT link0in:
  PLACE antichannel[32] AT link1in:
  PLACE antichannel[29] AT link2out:
  PLACE antichannel[31] AT link3out:
  corner.node(15,channel[29],channel[31],channel[30],
                            channel[32],antichannel[29],
     antichannel[31],antichannel[30],antichannel[32])

PROCESSOR 4 T4
  PLACE channel[2] AT link3out:
  PLACE channel[4] AT link0out:
  PLACE channel[5] AT link1out:
  PLACE channel[6] AT link2out:
  PLACE antichannel[2] AT link3in:
  PLACE antichannel[4] AT link0in:
  PLACE antichannel[5] AT link1in:
  PLACE antichannel[6] AT link2in:
  cross.node(4,antichannel[2],antichannel[4],
    antichannel[5],antichannel[6],channel[2],
            channel[4],channel[5],channel[6])
```

```
PROCESSOR 6 T4
  PLACE channel[18] AT link3out:
  PLACE channel[14] AT link0out:
  PLACE channel[20] AT link1out:
  PLACE channel[21] AT link2out:
  PLACE antichannel[18] AT link3in:
  PLACE antichannel[14] AT link0in:
  PLACE antichannel[20] AT link1in:
  PLACE antichannel[21] AT link2in:
  central.node(6,antichannel[18],antichannel[14],
         antichannel[20],antichannel[21],channel[18],
         channel[14],channel[20],channel[21])

PROCESSOR 12 T4
  PLACE channel[32] AT link0in:
  PLACE antichannel[0] AT link1in:
  PLACE antichannel[27] AT link2in:
  PLACE antichannel[8] AT link3in:
  PLACE antichannel[32] AT link0out:
  PLACE channel[0] AT link1out:
  PLACE channel[27] AT link2out:
  PLACE channel[8] AT link3out:
  corner.node(12,antichannel[8],channel[32],
  antichannel[0],antichannel[27],channel[8],
     antichannel[32],channel[0],channel[27])

PROCESSOR 14 T4
  PLACE channel[28] AT link0in:
  PLACE antichannel[17] AT link1in:
  PLACE antichannel[31] AT link2in:
  PLACE antichannel[22] AT link3in:
  PLACE antichannel[28] AT link0out:
  PLACE channel[17] AT link1out:
  PLACE channel[31] AT link2out:
  PLACE channel[22] AT link3out:
  cross.node(14,antichannel[22],channel[28],
             antichannel[17],antichannel[31],
                 channel[22],antichannel[28],
                     channel[17],channel[31])
```

APPENDIX E

EXPANDABLE CHANNEL PLACEMENT

```
{{{
{{{ define link/channel numbers - T4
VAL link0out  IS   0:
VAL link1out  IS   1:
VAL link2out  IS   2:
VAL link3out  IS   3:
VAL link0in   IS   4:
VAL link1in   IS   5:
VAL link2in   IS   6:
Val link3in   IS   7:
}}}
{{{  create internal mapping arrays
VAL  left.to.right.in    IS  [link0in,  link3in,   link1in,
link2in] :
VAL  right.to.left.in    IS  [link2in,  link1in,   link3in,
link0in] :
VAL  top.to.bottom.in    IS  [link1in,  link0in,   link2in,
link3in] :
VAL  bottom,to,top,in    IS  [link3in,  link2in,   link0in,
link1in] :
VAL                    left.to.right.out                    IS
[link2out,link1out,link3out,link0out ] :
VAL                    right.to.left.out                    IS
[link0out,link3out,link1out,link2out ] :
VAL                    top.to.bottom.out                    IS
[link3out,link2out,link0out,link1out ] :
VAL                    bottom.to.top.out                    IS
[link1out,link0out,link2out,link3out ] :

-- each soft channel is associated  with  a  table  which is
indexed
-- when the soft channel is placed on to a hard channel.
}}}
{{{  declare size structure
VAL  n  IS  4:
VAL  p  IS  n:             -- x dimension of array
VAL  q  IS  n:             -- y dimension of array
VAL  nodes IS p * q:
}}}
{{{   declare size channels
[nodes]       CHAN left.to.right,
                   right.to.left:
[nodes + 1]   CHAN top.to.bottom,
                   bottom.to.top:
}}}
```

```
{{{  node 1
{{{  declaration of constants
VAL i IS 0:
VAL j IS 0:
VAL dec.machine  IS 0:
VAL left         IS (dec.machine + (nodes - q)) \ nodes:
VAL right        IS dec.machine:
VAL bottom       IS dec.machine:
VAL top          IS nodes:
VAL map.index    IS ((j\2)*2) + (i\2):
}}}
PROCESSOR 1 T4
  {{{ placement of channels
  PLACE   left.to.right      [left]     AT  left.to.right.in
[map.index]:
  PLACE   left.to.right      [right]   AT  left.to.right.out
[map.index]:
  PLACE   right.to.left      [right]   AT   right.to.left.in
[map.index]:
  PLACE  right.to.left     [left]      AT  right.to.left.out
[map.index]:
  PLACE   top.to.bottom      [top]      AT  top.to.bottom.in
[map.index]:
  PLACE  top.to.bottom      [bottom]   AT  top.to.bottom.out
[map.index]:
  PLACE   bottom.to.top      [bottom]   AT  bottom.to.top.in
[map.index]:
  PLACE    bottom.to.top          [top]                    AT
bottom.to.top.out[map.index]:
  }}}
  node (1, left.to.right [left],left.to.right [right],
        right.to.left [right], right.to.left [left],
        top.to.bottom [top], top.to.bottom [bottom],
        bottom.to.top [bottom], bottom.to.top [top] )
  }}}

  {{{  node q
  {{{  declaration of constants
  VAL i IS 0:
  VAL j IS q-1:
  VAL dec.machine  IS q-1:
  VAL left         IS (dec.machine + (nodes - q)) \ nodes:
  VAL right        IS dec.machine:
  VAL bottom       IS dec.machine:
  VAL dec.j        IS (j + (q-1)) \ q:
  VAL top          IS dec.j + (i * q):
  VAL map.index    IS ((j\2)*2) + (i\2):
  }}}
  PROCESSOR q T4
  {{{ placement of channels
  PLACE   left.to.right      [left]     AT  left.to.right.in
[map.index]:
  PLACE   left.to.right      [right]   AT  left.to.right.out
[map.index]:
  PLACE   right.to.left      [right]   AT   right.to.left.in
[map.index]:
  PLACE  right.to.left    [left]      AT  right.to.left.out
[map.index]:
  PLACE   top.to.bottom      [top]      AT  top.to.bottom.in
[map.index]:
  PLACE  top.to.bottom      [bottom]   AT  top.to.bottom.out
[map.index]:
  PLACE   bottom.to.top      [bottom]   AT  bottom.to.top.in
[map.index]:
  PLACE    bottom.to.top          [top]                       AT
bottom.to.top.out[map.index]:
  }}}
  node (q, left.to.right [left],left.to.right [right],
        right.to.left [right], right.to.left [left],
        top.to.bottom [top], top.to.bottom [bottom],
        bottom.to.top [bottom], bottom.to.top [top] )
  }}}
  VAL i  IS 0:
  PLACED PAR J = 1 for (q-2)
    VAL dec.machine  IS j + (i * q) :
    VAL machine  IS dec.machine + 1 :


    PROCESSOR machine T4
      {{{    evaluate indices
      VAL left         IS (dec.machine + (nodes-q)) \ nodes:
      VAL right        IS dec.machine:
      VAL bottom       IS dec.machine:
      VAL dec.j        IS (j + (q-1)) \ q:
      VAL top          IS dec.j + (i * q) :
      VAL map.index    IS ((j\2) * 2) + (i\2) :

                    -- position  of  node  within  the  B003
group.

      }}}
  {{{ placement of channels
  PLACE   left.to.right      [left]     AT  left.to.right.in
[map.index]:
  PLACE   left.to.right      [right]   AT  left.to.right.out
[map.index]:
  PLACE   right.to.left      [right]   AT   right.to.left.in
[map.index]:
  PLACE  right.to.left     [left]      AT  right.to.left.out
[map.index]:
  PLACE   top.to.bottom      [top]      AT  top.to.bottom.in
[map.index]:
  PLACE  top.to.bottom      [bottom]   AT  top.to.bottom.out
[map.index]:
  PLACE   bottom.to.top      [bottom]   AT  bottom.to.top.in
[map.index]:
  PLACE    bottom.to.top          [top]                    AT
bottom.to.top.out[map.index]:
  }}}
  node (machine, left.to.right [left],left.to.right [right],
        right.to.left [right], right.to.left [left],
        top.to.bottom [top], top.to.bottom [bottom],
        bottom.to.top [bottom], bottom.to.top [top] )

PLACED PAR i = 1 FOR (p - 1)
  PLACED PAR j = 0 FOR Q
    VAL dec.machine  IS j + (i * q) :
    VAL machine IS dec.machine + 1 :
    PROCESSOR machine T4
      {{{     evaluate indices
      VAL left         IS (dec.machine + (nodes-q)) \ nodes:
      VAL right        IS dec.machine:
      VAL bottom       IS dec.machine:
      VAL dec.j        IS (j + (q-1)) \ q:
      VAL top          IS dec.j + (i * q) :
      VAL map.index    IS ((j\2) * 2) + (i\2) :

                        -- position  of  node  within  the  B003
group.
      }}}
  {{{ placement of channels
  PLACE  left.to.right      [left]      AT  left.to.right.in
[map.index]:
  PLACE   left.to.right      [right]   AT  left.to.right.out
[map.index]:
  PLACE   right.to.left      [right]   AT   right.to.left.in
[map.index]:
  PLACE   right.to.left      [left]    AT  right.to.left.out
[map.index]:
  PLACE  top.to.bottom    [top]         AT  top.to.bottom.in
[map.index]:
  PLACE   top.to.bottom      [bottom]  AT  top.to.bottom.out
[map.index]:
  PLACE   bottom.to.top      [bottom]   AT  bottom.to.top.in
[map.index]:
  PLACE    bottom.to.top          [top]                    AT
bottom.to.top.out[map.index]:
  }}}
  node (machine, left.to.right [left],left.to.right [right],
        right.to.left [right], right.to.left [left],
        top.to.bottom [top], top.to.bottom [bottom],
        bottom.to.top [bottom], bottom.to.top [top] )

}}}
```

In this appendix we start the placement from processor 01 on.

The placement of channels in the I/O handler is as follows:

```
{{{
CHAN OF ANY leftin,rightout,antirightout,antileftin:
PLACE leftin AT link3in:
PLACE rightout AT link3out:
PLACE antirightout AT link2out:
PLACE antileftin AT link2in:
}}}
```

LIST OF REFERENCES

[AMES77] Ames, W.F., Numerical Methods, 2nd edition page 251, formula 5.64, Academic Press, NY, 1977.

[CAWE80] Weitzman Cay., Distributed Micro/Minicomputer Systems, Prentice-Hall,Inc, 1980.

[DASP78] Aspinall, D., The Microprocessor and its applications, Cambridge University Press, 1978.

[HOMO87] Howe, Carl and Moxon, Bruce., "How to program parallel processors", IEEE Spectrum, September, 1987.

[INMOSD86] Inmos., "IMS T414 Transputer", December, 1986.

[INMOSJ88] Inmos., "Introduction to Transputers", January, 1988.

[INMOSO86] Inmos., Reference Manual "Transputer", October, 1986.

[INMOSTN11] Inmos., "Technical Note Number 11"

[INMOSTN13] Inmos., "Technical Note Number 13"

[KAFA84] Hwang, Kai and Briggs, Faye., Computer Architecture and Parallel Processing, Mc Graw-Hill, 1984.

[KOD88] Kodres, Uno.,"MultiTransputer Network Programming with Shared Global Distributed Variables", Umpublished paper, Naval Postgraduate School, Monterey, California, 1988.

[LIMI87] Lipovski, G.J and Miroslaw Malek., PARALLEL COMPUTING Theory and Comparisons, Wiley-Interscience Publication, 1987.

[SIHA88] Hart, Simon., Design, Implementation and Evaluation of a Virtual Shared Memory System in a MultiTransputer Network, Naval Postgraduate School, Monterey, California, December, 1987.

[REKA79] Reed, David and Kanodia, Rajendra., "Synchronization with Eventcounts and Sequencers", Communication of the ACM, February,1979.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	02
2.	Library, Code 0142 Naval Postgraduated School Monterey, CA 93943-5002	02
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduated School Monterey, CA 93943	01
4.	Dr. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduated School Monterey, CA 93943	03
5.	Major Richard A. Adams, USAF, Code 52Ad Department of Computer Science Naval Postgraduate School Monterey, CA 93943	01
6.	Daniel Green, Code 20F Naval Surface Weapons Center Dahlgren, VA 22449	01
7.	Jerry Gaston, Code N24 Naval Surface Weapons Center Dahlgren, VA 22449	01
8.	Captain J. Hood, USN PMS 400b5 Naval Sea Systems Command Washington, DC 20362	01

9.	RCA AEGIS Repository RCA Corporation Government Systems Division Mail Stop 127-327 Moorestown, NJ 08057	01
10.	Library (Code E33-05) Naval Surface Weapons Center Dahlgren, VA 22449	01
11.	Dr. M. J. Gralia Applied Physic Laboratory John Hopkins Road Laurel, MD 20702	01
12.	Dana Small, Code 8242 Naval Ocean Systems Center San Diego, CA 92152	01
13.	Director de Mantenimiento de Sistemas de Armas Comandancia General de la Armada Ave. Vollmer, San Bernardino, Caracas Distrito Federal, Armada-001-MSA, Venezuela, South America	01
14.	Director de Educacion Comandancia General de la Armada Ave. Vollmer, San Bernardino, Caracas Distrito Federal, Armada-001-EED, VENEZUELA, South America	01
15.	Lieutenant Comander Jose Frazao Sosa Venezuelan Navy Agregaduria Naval de Venezuela 2409 California Street. N.W. Washington. DC. 20008	05
16.	Superintendent, Naval Postgraduate School Computer Technology Programs, Code 37 Monterey, CA 93943-5000	01